FINAL REPORT

February 28, 2009

AFOSR Grant FA9550-06-1-0164

DEVELOPMENT OF IMPLICIT COMPACT METHODS FOR CHEMICALLY REACTING FLOWS

Mitchell D. Smooke

Yale University
Department of Mechanical Engineering
Becton Center
15 Prospect Street
New Haven, CT 06511

AFRL-SR-AR-TR-09-0079

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
26-02-2009	Final	Mar 2006 - Nov 2008

4. TITLE AND SUBTITLE

Development of Implicit Compact Algorithms with Application to Chemically Reacting Flows

5a. CONTRACT NUMBER

5b. GRANT NUMBER

FA9550-06-1-0164

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Smooke, Mitchell
Long, Marshall

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Yale University, Grant & Contract Administration
155 Whitney Avenue, Suite 214
New Haven, CT 06520

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
875 N Randolph St
Arlington, VA 22203
Dr. Fariba Fahroo/NL

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION A: Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This project has focused on the design of numerical algorithms that are well suited to the computation of time-dependent chemically reacting flows with finite-rate kinetics and detailed transport. High order compact finite differences have been used to discretize the spatial operators since the spectral-like resolution of the small scales makes it feasible to conduct accurate, long-time computations of multidimensional flames burning real fuels. In view of the stiffness of the chemical mechanisms characterizing these fuels, implicit time integration techniques have been employed. The fully coupled implicit-compact solver developed during this grant has been successfully applied to a sequence of test problems, from convection-diffusion equations with analytical solutions to multicomponent low-speed heated jet flows in two dimensions to a model premixed flame with two-step Arrhenius chemistry. Along the way, important advances have been made in the following areas: highly efficient algorithms for storing and manipulating the Jacobian matrix in the Newton solver, robust preconditioned iterative linear algebra methods, strategies for steady solutions with high order discretizations, and automated code development tools. Current work is devoted to applying the implicit compact solver

15. SUBJECT TERMS

high-order discretizations, compact methods, implicit time stepping, combustion, diagnostics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*

Introduction

It is clear from both a logistical and an economic viewpoint that the combustion of hydrocarbon fuels will continue to play a central role in US Air Force operations. In addition, with world events demanding enhanced flexibility in the sources of fuels, it is becoming increasingly important to improve scientific knowledge of the combustion properties of different fuel blends. Chemically reacting flows utilizing hydrocarbon fuel blends occur in a variety of energy conversion processes such as combustion, propulsion, and fuel reforming, to name just a few. They are also relevant to many material synthesis technologies. If the chemical transformations and, in some cases, attendant energy release can be made to happen in a well-controlled and well-defined fashion, the goal of either liberating heat, partially oxidizing a fuel, generating electrical power, or synthesizing advanced materials should be achievable with high efficiency and minimal pollution.

Hydrocarbon fuels will remain a major source of energy well into the second half of the 21st century and, despite dire warnings about their limited supply, known resources have actually increased over the past decade. Nevertheless, finite supplies will continue to exert pressure on the efficient use of these fuels, especially as the price of oil continues to skyrocket. In the engineering of chemically reacting flows, increased efficiency and reduced pollution can be achieved via an integrated approach that extends the state-of-the-art of both experimental and computational methodologies. By making advances in each of these areas and by integrating them in well-conceived research programs, scientists will be able to have a dramatic impact on the design of technologies involving energy conversion and combustion. One of the most technically challenging engineered systems of importance to the Air Force in which chemically reacting flows play a critical role are gas turbines (GTs). For aeropropulsion applications, there are no alternative energy replacements of GTs in sight. In addition, the majority of the electrical power to be added in the United States and around the world through 2015 will be based on GTs. Advances in GT engineering would inevitably affect the entire combustion industry, and the economic repercussions of these advances would be amplified further as critical combustion issues for GTs are also important to the transportation industry. Moreover, the economic payoff for US Military Operations could be enormous.

Many contributions are needed to advance the frontiers of the science behind such engineered systems, and thereby to enhance the nation's economic base and help stabilize it against foreign competition and dependencies. The research will focus on the development of advanced computational methodologies that will enable reacting flow simulations which are more rapid and more accurate than those currently feasible, and which, on both counts, are capable of deepening an understanding of the fluid dynamics and aerothermochemistry underlying many vital technologies. Specifically, this research has considered numerical algorithms designed for the solution of gas-phase combustion with detailed transport and finite-rate chemistry. To help achieve these goals, a companion experimental program has been initiated in which the complexity of the various systems is being dissected into well-defined laboratory-scale problems, from which data can be provided for the validation of the computational models.

Overview of the Implicit-Compact Solver

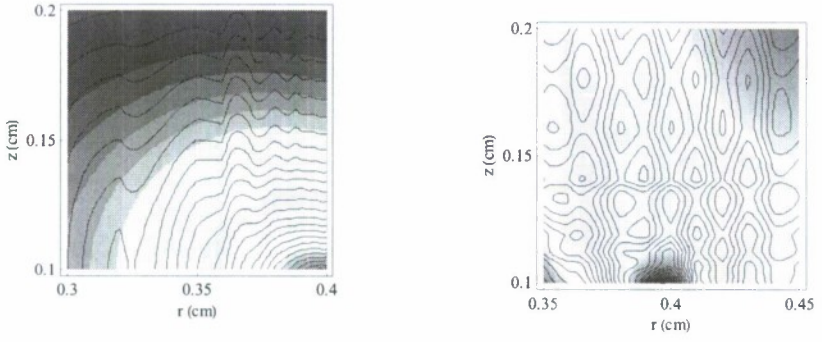
The implicit-compact methods studied in this granting period have been designed to meet two well-known challenges in modeling time-dependent combustion: the stiffness induced by the vastly disparate timescales in the chemistry, which calls for implicit time integration; and the significant spatial structure in the flow field, which cannot be captured without high resolution, low diffusivity spatial discretizations. The main idea of a compact scheme discretization is to construct algebraic relationships between the values of a function and of its derivative at the nodes of a grid. These equations are written in matrix form; the matrices are banded, generally tridiagonal, and hence can be inverted efficiently. The coefficients are constant for a given grid and are defined by matching the terms of Taylor series expansions. The spatial discretizations used in this work have a variable order of accuracy, depending on the grid spacing and the presence of steep gradients near the domain boundaries. The maximum order is six and the minimum is three. Quite apart from their classical order of accuracy, the particular advantage of the compact schemes is their "spectral-like" resolution of moderately high wave numbers. In practice, this allows good accuracy over a long time with many fewer grid points than would be required by a traditional low order finite difference method.

In the implicit-compact solver, the governing partial differential equations (PDEs) are semi-discretized using a compact finite difference procedure (a finite volume method could be used too). Then, after the spatial discretization, the system of ordinary differential equations is discretized with an A-stable backward difference formula (BDF), following the "method of lines" approach. The resulting nonlinear algebraic system is solved by a damped inexact Newton's method. An approximate solution to the linearized Newton system is obtained using an iterative Krylov method (GMRES) with an appropriate preconditioner (incomplete LU decomposition with a scaling/reordering preprocessor). The solver has been thoroughly tested on problems with known analytical solutions, thus verifying the correctness of all temporal and spatial discretizations. Seeing as the preconditioner is the most expensive part of the solution process, coarse-grain parallelism was introduced into this module by means of restricted additive Schwarz domain decomposition, implemented in a shared memory context by OpenMP pragmas. The payoff here was not significant (due to memory bandwidth issues, the code could only run effectively on 4-6 threads at once), and the extension of the domain decomposition algorithm to a distributed memory context remains a task for future work. As a result of an AFOSR DURIP award, the hardware for this work is already in place: a cluster with 128 cores, 512 GB of RAM, and several terabytes of disk, all connected via a high-speed DDR Infiniband fabric.

The code has been written in Fortran. By now, almost all of the modules have been (at least partly) modernized to take advantage of Fortran 90/95 enhancements in the area of array processing, user-defined data structures, and data encapsulation in modules. The only exception is the MC64 code, which has been taken from the Harwell Subroutine Library. The Fortran for all of the problem-specific subroutines has been produced by a *Mathematica* code generation tool, as noted below. Partial listings of this script and of the residual and Jacobian subroutines created with it are attached as Appendices.

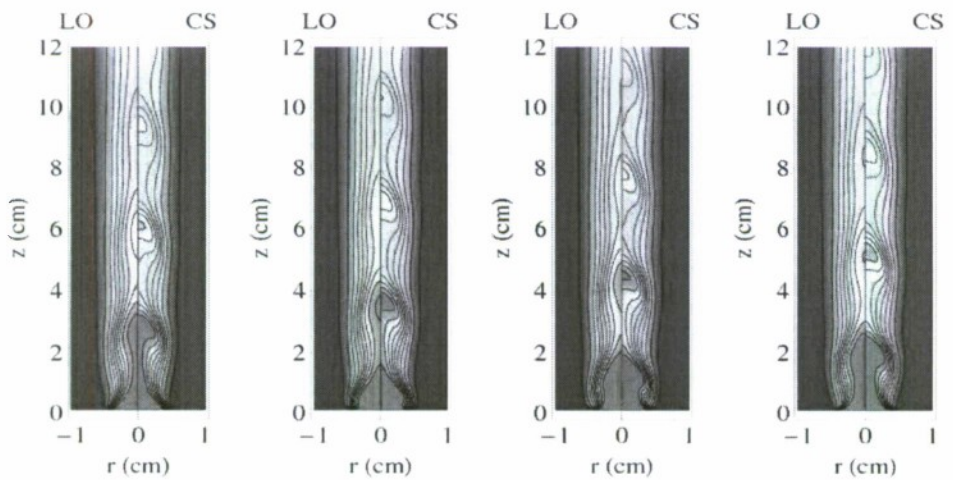
Accomplishments: Modeling

(a) *Computing the pressure field*: Originally, the plan was to work with a velocity-vorticity formulation of the fluid dynamical problem, a decision based primarily on the experience obtained in using vorticity-based methods to model flames. A side benefit here was that there would be no need to contend with the signature challenge of low speed flows: computing the pressure field. Before long, however, it became clear that the velocity-vorticity formulation leads to intractable linear systems in the Newton iteration for time-dependent problems discretized in space with compact schemes. Accordingly, during the first six months of the grant, the intended approach to the fluid dynamics was abandoned in favor of a primitive variables formulation. Since acoustic effects were not of primary interest, the zero Mach number approximation was employed in formulating the governing equations. As is standard in the modeling of low-speed flows, the hydrodynamic pressure was taken as the independent variable, thermodynamic pressure was constant, and the density was recovered from the ideal gas law. Note that, with a fully coupled solver, continuity can be enforced even though dynamic pressure is retained as the corresponding unknown. Also, by constructing the numerical method to take advantage of the natural coupling of the variables, a Poisson equation for pressure and a pressure projection step are not needed. In studying flow in a pipe, it was initially observed that the smoothness of the solutions tended to be very sensitive to the choice of grid; specifically, it was found that local under-resolution of the radial velocity field led to noticeable oscillations in the pressure field along most of the length of the pipe. The lesson learned was that the tight, global coupling of all unknowns in the implicit-compact discretization places high demands on the adequate spatial resolution of all structures and boundary layers in the flow field. Once this was achieved, the solver delivered strikingly smooth, accurate solutions, the likes of which it was impossible to achieve with traditional low order methods. These results have demonstrated that the notorious "pressure-velocity decoupling" problem, which manifests itself as an oscillatory pressure field in finite difference calculations using collocated grids, can be overcome by using compact schemes to discretize in space:



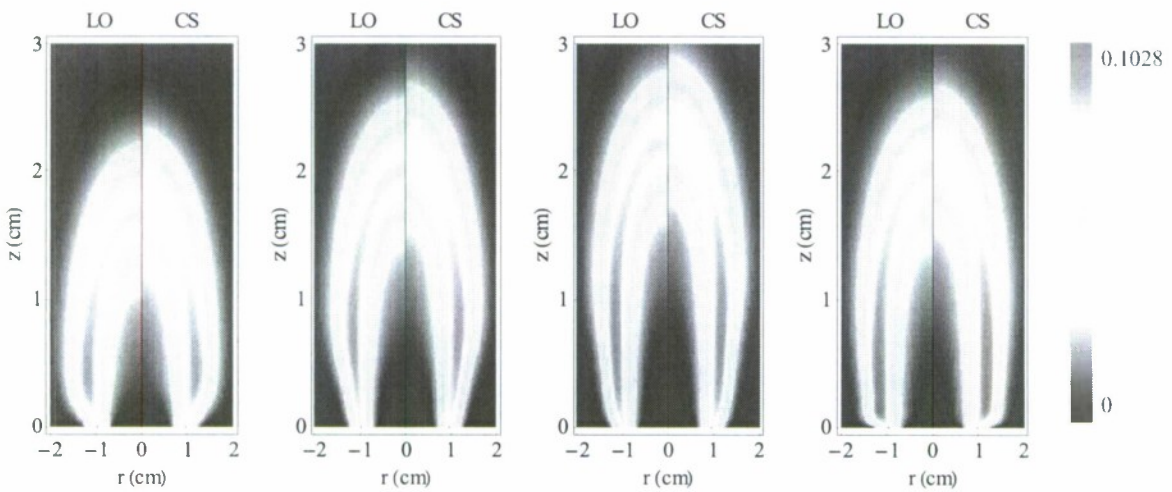
Contours of dynamic pressure, compact scheme (color) vs. low order method (black lines). Left: steady pipe flow (Re=500, pipe radius=0.4cm). Right: oscillating cold jet flow at 0.025 s. The small-amplitude grid-scale noise in the low order solutions is due to the pressure-velocity decoupling.

(b) *Unsteady multicomponent flows with thermal mixing*: Before attempting to compute reacting flows it was imperative to move beyond simple flows and demonstrate the capability of the implicit-compact solver to model complicated multicomponent flows with thermal and species mixing. The work focused on jet flows in quasi-open axisymmetric geometries, where the flow configuration was chosen to mimic that of the diffusion flames which are the goal of this research. Here, "quasi-open" means that a solid wall was placed in the radial far-field of the computational domain, at a distance of many jet radii from the centerline. Though this introduced a flow recirculation zone near this wall, the velocities were very small and any difficulty of computing the recirculation was more than compensated by the ability to use a simple "no slip" Dirichlet boundary condition there for the velocity field. This took the complicated issue of open boundary conditions off the table where it was most likely to be a sticking point, in the radial far-field (homogeneous Neumann outflow conditions are probably feasible for the high order solver since the grids used in flame calculations are typically very long in the axial direction). Calculations of an oscillating heated jet issuing into quiescent cool air were undertaken. The reference calculations used the same low order discretization employed in a previously developed flame code, namely, second order centered differences for diffusive terms and first order (monotone) upwinding for convective terms. These solutions were ruined by strong artificial viscosity from the upwind differencing. By contrast, the calculations with the compact scheme discretization successfully captured the spatial structure of the flow, revealing a marked "pinching" in the temperature field that was qualitatively similar to the thermal structure of the time-varying diffusion flame studied at Yale in the laboratory of Marshall Long. Both calculations were run with a time step small enough to ensure that the spatial error dominated. A comparison of the results is presented in the figure below.



Temperature contours in a forced, heated jet flow (Re=500, ΔT=100K), compact scheme (CS) vs. low order (LO) solution. The forcing frequency is 20 Hz. The images are taken at 0.0, 0.0125, 0.025, and 0.0375 s.

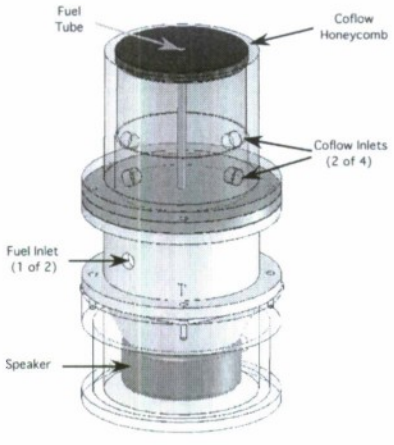
(c) *Complex chemistry capability*: A model premixed flame was studied in order to isolate the numerical challenges of realistic combustion thermochemistry from those relating to the fluid dynamics (the flow field is imposed in the model). Three convection-diffusion-reaction equations with exponentially nonlinear two-step Arrhenius chemistry were solved for temperature and two reacting species in a two-dimensional axisymmetric geometry. These calculations have provided a proof-of-concept that the implicit-compact methods can deliver accurate and efficient solutions to stiff multi-step chemistry combustion problems. These types of problems arise in modeling the combustion of aviation fuels of interest to the Air Force, and they create significant numerical challenges for both explicit methods (CFL restrictions) and many splitting methods (additional accuracy limitations due to splitting errors). Calculations of the steady, two-dimensional model flame were performed with both the compact scheme and the low order semi-discretizations, and the steady solutions were then used to initialize time-dependent calculations, where the imposed flow was periodically varying (much like in the oscillating jet flow discussed above). In the transient simulation, comparison of the compact scheme solution with the low order solution revealed the presence of numerical diffusion in the latter. This can be observed most clearly in the damping of the temporal oscillation of the intermediate species. This artifact of the spatial discretization had hampered earlier low order simulations of time-varying diffusion flames. The implicit-compact solver succeeds in generating spatially accurate solutions for such problems, and since it allows for large time steps the computational cost is reasonable.



Evolution of intermediate species concentration in the model premixed flame problem, compact scheme (CS) vs. low order (LO) solution. The frequency of oscillation is 10 Hz. The images are taken at 0.0, 0.02, 0.04, and 0.06 s, respectively.

(d) *Experimental validation of the solver*: Concurrent with the numerical development was the construction of the experimental configuration for validating the new implicit-compact solver, which will be essential as the work transitions to detailed chemistry flame simulations. Toward the end of the granting period, a burner was tested in which

fuel flows from a 0.4 cm inner diameter vertical tube (wall thickness 0.038 cm) into a concentric, 7.4 cm diameter oxidizer coflow.



CAD drawing of the forced-flow burner used in the time-varying diffusion flame experiments.

A speaker in the plenum of the fuel jet allows a periodic perturbation to be imposed on the exit parabolic velocity profile. The fuel is diluted with nitrogen and the velocities have been carefully tuned so as to produce a flame with negligible soot that is lifted above the burner surface, preventing heat transfer from the flame to the burner. These operating conditions have been designed to simplify the calculations in such a way that physical realism is not sacrificed. Recently, laser diagnostic techniques such as Raman and Rayleigh scattering have been used to measure temperature and major species profiles from flames using this burner.

Accomplishments: Numerical Methods and Software

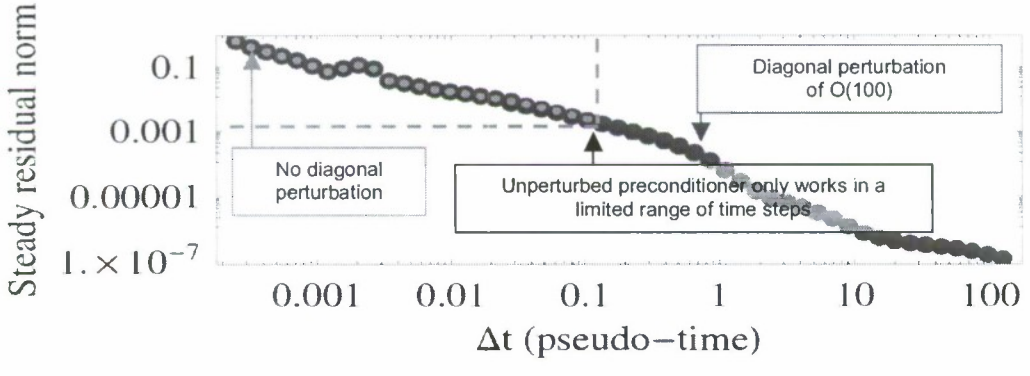
(a) *Efficient Jacobian operations for compact scheme discretizations*: The first hurdle in developing the implicit-compact methodology has been to devise a set of highly efficient numerical algorithms for the formation, multiplication, and element-extraction of the Jacobian matrix, and to develop the means to implement these algorithms effectively in a large scientific code. The methods used to accomplish these tasks have come to maturity over the past few years. Though dense by any typical criterion of matrix sparsity, compact scheme Jacobians possess a latent structure based on the fact that they arise from a discretization of a PDE. The idea of decomposing the Jacobian into "local" and "spatial" components exploits this structure to achieve a form of data compression. To illustrate the idea: if U is the independent variable and F = F(U,Ux) is the residual, which is a function of U and its spatial derivative Ux, the Jacobian matrix in Newton's method can be reconstructed by considering Ux to be independent of U, forming the "local" Jacobian components dF/dU and the "spatial" Jacobian components dF/d(Ux), and observing that J = dF/dU + (dF/d(Ux))*Dx, where Dx is the appropriate coefficient of the differentiation matrix used to calculate Ux from U. The result of using this decomposition

is that, even for the most memory intensive combustion problems, a Jacobian arising from a compact scheme spatial discretization can be stored with *less* memory than a conventional finite difference method would need to store an equivalent low order Jacobian. Moreover, once generated, the compact scheme Jacobian can be applied to a vector extremely efficiently, since J*V = (dF/dU)*V + (dF/d(Ux))*(Dx*V) = (dF/dU)*V + (dF/d(Ux))*Vx. Here it is seen that J*V is equal to the sum of two dot products, one of the "local components" with the vector V and the other of the "spatial components" with the vector Vx. The fundamental operation in any iterative linear algebra routine is the matrix-vector product. With this algorithm it can be computed in O(N) flops.

(b) *Linear algebra enhancements I. Robust iterative methods*: A fully coupled solution paradigm makes little sense without robust and efficient numerical linear algebra algorithms. These already exist for the low order spatial discretizations commonly used in computational combustion; for the compact discretizations, they have been discovered and fine-tuned during the tenure of this grant. In the first year, in the course of studying pipe flows, it was found that the use of large grids (i.e., many points) led to stagnation in the preconditioned Bi-CGSTAB linear solver. The difficulties were more severe than anything encountered using Bi-CGSTAB in solving combustion problems within the past fifteen years. To overcome them, a new iterative linear algebra module was developed and integrated into the implicit-compact solver. Based on GMRES, it enjoys the monotonicity and enhanced robustness of this method, while retaining the state-of-the-art MC64-ILUT preconditioner that has proven effective for the challenging linear systems produced by compact semi-discretization. The implementation made some sacrifices of efficiency for greater reliability, such as eschewing restarts and performing modified Gram-Schmidt with full re-orthogonalization for the Arnoldi process, though the penalty incurred was minimal for time-dependent problems, where the Jacobian matrices have large diagonal terms and the linear system solution is fast. In any case, even in the absence of optimizations, the new linear solver not only made it possible to solve problems that had previously caused the code to fail (e.g., a large binary mixing problem on very nonuniform grids), but also displayed significant performance gains over its predecessor (e.g., showing between a two- and five-fold decrease in the time spent solving linear systems). A decade ago, memory constraints made the use of GMRES in flame calculations much more difficult than it is today.

(c) *Linear algebra enhancements II. Robust preconditioning*: The basic approach of applying a purely algebraic MC64-ILUT preconditioning algorithm to a pre-sparsified or "partial" Jacobian matrix is sound, but also potentially expensive and difficult to optimize due to the parameters in the algorithm. Early on, by performing a sequence of tests on basic fluid dynamics test problems one could sample enough of the parameter space of the preconditioner to come to a satisfactory understanding of how at least to get the incomplete factorization to work. With ILUT, it became clear that the drop tolerance was the more expensive and less effective parameter to tune. Hence, the strategy originally employed was to compute with a modest fill-in parameter and to recompute with more fill if the linear solver failed. As long as this worked, it was fine; however, as the difficulty of the physical models and the size of the problems were scaled up, both the heightened importance of the preliminary sparsification phase and the pressure, for the sake of computational efficiency, to form the preconditioner less frequently clarified a

new question: when the linear solver fails, is this due to a specific problem with the ILU or rather simply to an out-of-date or otherwise ineffective preconditioner? In the effort to answer this question, the linear solver module was enhanced by the addition of a number of inexpensive sanity checks that helped us to interpret the results of successful linear solves and to diagnose the cause of failures. By far the most frequent cause of failure was instability in the ILU. This was deduced by comparing the condition estimates of the ILU and of the preconditioned linear system, or rather, of the Hessenberg matrix constructed by GMRES, which represents the projection of this linear system onto the orthonormalized Krylov basis. A poorly conditioned ILU indicates instability in the incomplete factorization process; given a well conditioned ILU, a poorly conditioned Hessenberg suggests that the preconditioner is inaccurate. The diagnosis is straightforward, and in the former case, so is the cure: perturb the diagonal of the partial Jacobian and recomputed the ILU. This understanding was a breakthrough for two reasons. First, it meant that many, perhaps the majority of, failed linear solves could be rectified at essentially no increase in computational cost. Applying a diagonal perturbation is a negligible expense compared to the ILU, the ILU can be performed without increasing the level of fill-in, and if the perturbation is not large the resulting preconditioned Krylov process can still converge in a reasonable number of iterations. By contrast, the previous approach to addressing failure in the linear solver (recomputing the ILU with more fill-in) has a cost in terms of both memory usage and time which grows unpredictably with increasing fill. Second, the use of diagonal perturbations has allowed stabilized ILU factorizations of compact scheme partial Jacobians at *much* larger time steps than otherwise possible. This has again made it feasible to calculate steady solutions with the compact scheme solver for some large-scale problems where previously this had been out of the question (see the figure below, and discussion in (d)).



(d) *Constructing good initial conditions for time-dependent problems*: The semi-discretized governing equations for a transient flame problem in the primitive variable formulation are a system of differential-algebraic equations (DAEs). As is well known, a DAE requires consistent initialization: it is important to construct an initial condition that satisfies the algebraic constraints so as to avoid boundary layers and maintain accuracy in the early phase of the computations. Moreover, if high order spatial discretizations were used in moving from the original PDE to the DAE, this initial condition must itself meet strict smoothness requirements. A basic challenge for those employing high order

methods is how to generate such an initial condition. In practice, one often wishes to use a steady-state solution for this purpose. This was the approach assumed at the inception of this research program. However, it was very difficult to generate a steady compact scheme solution from scratch using Newton iteration with pseudo-transient continuation, the method of choice for problems discretized with traditional low order finite differences. The source of the difficulty is that, contrary to their name and their reputation, the *effective* computational stencils of compact schemes are very wide (the inverse of a tridiagonal matrix is structurally dense), so Jacobian matrices based on compact scheme discretizations can be, and at steady state generally are, very far from diagonal dominance, and hence very difficult to precondition using known methods. As a workaround, a low-cost and effective means of computing satisfactorily smooth and accurate initial conditions for high order flow simulations was developed. The method takes the "inadequate" steady-state solutions produced by low order solvers and marches them in pseudo-time with moderately large time steps using the implicit-compact solver. This procedure has been greatly facilitated by progress in learning how to stabilize the ILU preconditioner at such time steps, as described above.

(e) *Code generator*: At present, the entire computational kernel of the implicit-compact solver (the residual function, the Jacobian operations, and all of the problem-specific preconditioner routines) can be generated automatically in Fortran or C, by simply entering the partial differential equations and the boundary conditions in standard mathematical notation and executing a *Mathematica* script. For simplicity, vector calculus operators (Grad, Div) can be used in formulating the problem, which another freely available *Mathematica* package translates into the correct partial derivatives based on the geometry specified (Cartesian, Cylindrical, etc.). This software tool has drastically shortened the programming and debugging phases of code development, and after reaching its current mature state it has allowed the realization of a relatively fast turnaround on the study of new problems/problem formulations. In principle, this code generation script could be turned into a *Mathematica* package and made available to any researcher who employs finite difference methods to solve partial differential equations. As noted in the Overview section of this report, a partial listing of the script is attached as an Appendix to this report.

Personnel Supported During Duration of Grant

Mitchell D. Smooke, Professor
Marshall B. Long, Professor
Richard Dobbins, Graduate Student

Publications

R.R. Dobbins and M.D. Smooke, "Implicit-Compact Methods for Systems of Convection-Diffusion Equations," in preparation.

R.R. Dobbins and M.D. Smooke, "Development of Implicit-Compact Methods for Finite-Rate Chemistry Combustion Problems," in preparation.

R.R. Dobbins and M.D. Smooke, "Initializing Transient Flow Simulations with the Implicit-Compact Solver: The Role of Conventional Low Order Methods," in preparation.

Honors and Awards Received

Program Chair, 32rd International Combustion Symposium

Chair, Connecticut Academy of Science and Engineering Transportation Systems Technical Board

Member of the External Advisory Board, Department of Mechanical Engineering, University of Connecticut

Member of the Engineering Advisory Board, Fairfield University

Member of the Board of Directors, The Combustion Institute

AFRL Point of Contact

Dr. Tim Edwards, AFRL/PRTG, Wright-Patterson AFB, OH, Phone 937-255-3524. Last meeting occurred at the AIAA ASM in Reno, Nevada, January 2008.

Transitions

None

APPENDIX I:

Mathematica-based Code Generation Tool

(partial code listing)

Primitive variable formulation of the PDEs governing time-dependent binary mixing

Richard Dobbins - January 2009

Session

◇ *Working directory*

```
Directory[]
```

```
SetDirectory["C:\\Documents and Settings\\rd\\My Documents
   \\Work - Engineering\\- RESEARCH -\\PROJECTS\\Fluids\\MixingPipe"]
```

```
SetDirectory["F:\\PROJECTS\\Combustion\\OneStep_Diffusion"]
```

```
F:\PROJECTS\Combustion\OneStep_Diffusion
```

◇ *Miscellaneous*

```
<< Format.m
```

```
Make sure that the package FortranDformat.m is not already loaded, or else
  FortranAssign's formatting of some numbers as DP constants will not work correctly!
```

```
Off[General::"spell1"]
Off[General::"spell"]
NormalPageWidth = PageWidth /. Options[$Output, PageWidth];
NotZeroQ[X_] := Not[Developer`ZeroQ[X]];
```

◇ *Error messages*

```
problemsetup::notcomplete =
  "Basic arrays (e.g. variables, derivatives,
    assumptionsRules) are not yet defined. Must complete problem setup.";
```

◇ *PageWidth and linebreaking*

Definitions & initializations

■ Problem setup

◇ *Coordinate system*

```
<< Calculus`VectorAnalysis`
SetCoordinates[Cylindrical[r, θ, z]]
```

Using NEW VectorAnalysis package which handles tensor operations

```
Cylindrical[r, θ, z]
```

◇ *Variables & material properties*

```
pres = P[t, r, θ, z];
vr = U[t, r, θ, z];
vθ = V[t, r, θ, z];
vz = W[t, r, θ, z];
v = {vr, vθ, vz};
temp = T[t, r, θ, z];
yk = YK[t, r, θ, z];

ρ = RHO[pres, temp, yk];

cp = CP[temp];
λcp = LACP[temp]; (* λ/cp - a simple function of T *)
μ = PR λcp;        (* PR - Prandtl number *)
ρDk = 1/LEk λcp;   (* LEk - Lewis number of species k *)

vCr = UC[t, r, θ, z];
vCθ = VC[t, r, θ, z];
vCz = WC[t, r, θ, z];
vC = {vCr, vCθ, vCz};

ωdot = SOURCE[temp, yk];
qdot = HEAT[temp, yk];

gr = GR;
gθ = GTHETA;
gz = GZ;
g = {gr, gθ, gz};
```

◇ *Governing equations*

```
Π = μ (Grad[v⃗] + Transpose[Grad[v⃗]]) + (μB - 2/3 μ) Div[v⃗] IdentityMatrix[3] ;
(** deviatoric stress tensor **)

ContinuityEquation = ∂t ρ + Div[ρ v⃗];
NavierStokesEquation = ρ ∂t v⃗ + ρ v⃗.Grad[v⃗] + Grad[pres] - ρ g⃗ - Div[Π];
EnergyEquation =
  ρ ∂t temp + ρ v⃗.Grad[temp] - Div[λcp Grad[temp]] - λcp/cp Grad[cp].Grad[temp] - qdot/cp;
SpeciesConservation = ρ ∂t yk + ρ v⃗.Grad[yk] + Div[ρ yk v⃗C⃗] - Div[ρDk Grad[yk]] - ωdot;
```

◇ *Boundary conditions*

```
(* INLET *)
B1p = NavierStokesEquation[[3]];
B1u = vr;
B1w = vz - Winlet;
B1t = temp - Tinlet;
B1y = yk - YKinlet;

(* SYMMETRY *)
B2p = ∂r pres;
B2u = vr;
B2w = ∂r vz;
B2t = ∂r temp;
B2y = ∂r yk;

(* WALL *)
B3p = NavierStokesEquation[[1]];
B3u = vr;
B3w = vz;
B3t = temp - Twall;
B3y = -ρDk Grad[yk][[1]];
(* no diffusion of species k into the wall: Vk,r Yk = 0, using Fick's Law *)

(* OUTLET *)
B4p = pres - AtmPressure;
B4u = ∂z vr + vr;  (* Robin condition *)
B4w = ∂z vz;
B4t = ∂z temp;
B4y = ∂z yk;
```

◇ *"Rules" expressing physical assumptions*

◇ *Assumptions made in this problem*

* 2-D axisymmetric flow (no swirl)
* negligible bulk viscosity
* axial gravity

```
assumptionsRules =
  {
    (* steadyFlowRule, *)
    twoDimensionalAxisymmetricFlowRule,
    negligibleBulkViscosityRule,
    axialGravityRules
   } // Flatten;
```

◇ *Problem summary*

```
problem = "Time-dependent Axisymmetric Diffusion Flame with One-Step Chemistry";
equations = {"Continuity", "Radial Momentum",
   "Axial Momentum", "Temperature", "Species Conservation"};
equationnumbers = Range[Length[equations]];
boundaries = {"Inlet", "Axis of Symmetry", "Wall", "Outflow"};
variables = {P, U, W, T, YK};
species = {"CH4", "O2", "CO2", "H2O", "N2"};
nspecies = Length[species]; (* number of species YK *)
derivtypes = {r, z, rr, zz, rz};
nderivtypes = Length[derivtypes];
Outer[ StringJoin, Map[ToString, variables], Map[ToString, derivtypes] ] // Flatten;
derivatives = Map[ToExpression, %];

Print[
  "PROBLEM SUMMARY: ", problem, "\n",
  "Equations = ", equations, "\n",
  "Variables = ", variables, If[nspecies > 0, "\n", ""],
  If[nspecies > 0, "Species   = ", ""], If[nspecies > 0, species, ""]
 ];
```

```
PROBLEM SUMMARY: Time-dependent Axisymmetric Diffusion Flame with One-Step Chemistry
Equations = {Continuity, Radial Momentum, Axial Momentum, Temperature, Species Conservation}
Variables = {P, U, W, T, YK}
Species   = {CH4, O2, CO2, H2O, N2}
```

▪ Compact Scheme code generation tools

◇ *Translation from Mathematica expressions to code symbols*

```
derivativeTranslationRules =
  {

   Derivative[0, 1, 0, 0][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "r"],
   Derivative[0, 0, 1, 0][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "q"],
   Derivative[0, 0, 0, 1][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "z"],
   Derivative[0, 2, 0, 0][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "rr"],
   Derivative[0, 0, 2, 0][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "qq"],
   Derivative[0, 0, 0, 2][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "zz"],
   Derivative[0, 1, 0, 1][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "rz"],
   Derivative[0, 1, 1, 0][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "rq"],
   Derivative[0, 0, 1, 1][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "qz"],
   Derivative[1, 0, 0, 0][A_][t, r, θ, z] :→ Symbol[ToString[A] <> "t"],

   (* "p" = derivative w.r.t. ρ *)
   Derivative[1, 0, 0][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "p"], Derivative[0, 1, 0][A_][P[t, r, θ, z],
     T[t, r, θ, z], YK[t, r, θ, z]] :→ Symbol[ToString[A] <> "t"],
   Derivative[0, 0, 1][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "y"],
   Derivative[2, 0, 0][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "pp"],
   Derivative[0, 2, 0][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "tt"],
   Derivative[0, 0, 2][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "yy"],
   Derivative[1, 1, 0][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "pt"],
   Derivative[0, 1, 1][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "ty"],
   Derivative[1, 0, 1][A_][P[t, r, θ, z], T[t, r, θ, z], YK[t, r, θ, z]] :→
    Symbol[ToString[A] <> "py"],

   Derivative[1][A_][T[t, r, θ, z]] :→ Symbol[ToString[A] <> "t"],
   Derivative[2][A_][T[t, r, θ, z]] :→ Symbol[ToString[A] <> "tt"]

  };

toCodeNotation[pde_] :=
 (
    (
      Expand[pde] /. derivativeTranslationRules
    ) /. {A_[t, r, θ, z] → A}
  ) /. {A_[P, T, YK] → A, A_[T] → A} /. {A_[T, YK] → A}
```

◇ *Array pointers*

```
(* "pad" pointers with spaces where needed to improve code legibility *)

padPointers[ptrs_] :=
  Module[{ptrnames, newptrnames = {}, ptrvalues, nptrs, paddedlength, i, ptrinchars},
   ptrnames = ptrs[[All, 1]];
   ptrvalues = ptrs[[All, 2]];
   nptrs = Length[ptrnames];
   paddedlength = Max[StringLength /@ ptrnames];
   For[i = 1, i ≤ nptrs, i++,
    ptrinchars = Characters[ptrnames[[i]]];
    If[paddedlength > Length[ptrinchars],
     ptrinchars = PadRight[ptrinchars, paddedlength, " "], ];
    AppendTo[newptrnames, StringJoin[ptrinchars]];
   ];
   Return[Table[{newptrnames[[i]], ptrvalues[[i]]}, {i, nptrs}]];
  ];

Module[

  {
   i,
   indexHeadEqns = "Meqn",
   indexHeadVars = "Mvar",
   indexHeadDers = "Kder",
   indname,
   indindx
  },

  If[Not[VectorQ[equationnumbers]] ||
    Not[VectorQ[variables]] || Not[VectorQ[derivatives]],
   Message[problemsetup::notcomplete]; Abort[];, ];

  Clear[indicesEqns, indicesVars, indicesDers];

  indicesEqns = StringJoin[indexHeadEqns, #] & /@ (ToString /@ variables);
  indicesVars = StringJoin[indexHeadVars, #] & /@ (ToString /@ variables);
  indicesDers = StringJoin[indexHeadDers, #] & /@ (ToString /@ derivatives);

  indicesEqns = Table[{indicesEqns[[i]], i}, {i, Length[indicesEqns]}];
  indicesVars = Table[{indicesVars[[i]], i}, {i, Length[indicesVars]}];
  indicesDers = Table[{indicesDers[[i]], i}, {i, Length[indicesDers]}];

  If[nspecies > 1,
   {indname, indindx} = Position[indicesEqns, "MeqnYK"] // Flatten;
   indicesEqns = ReplacePart[
     indicesEqns,
     StringJoin["(/ (",
      ToString[indicesEqns[[indname, indindx + 1]]], "-1+k, k=1,NSP_P} /)"],
```

```
    {indname, indindx + 1}
   ];
  {indname, indindx} = Position[indicesVars, "MvarYK"] // Flatten;
  indicesVars = ReplacePart[
    indicesVars,
    StringJoin["(/ (",
     ToString[indicesVars[[indname, indindx + 1]]], "-1+k, k=1,NSP_P) /)"],
    {indname, indindx + 1}
   ];
  YKdernames = Select[indicesDers[[All, 1]], StringMatchQ[#, "*YK*"] &];
  indlist = Flatten[Position[indicesDers, #] & /@ YKdernames, 1];
  Do[
   {indname, indindx} = indlist[[k]];
   indicesDers = ReplacePart[
     indicesDers,
     StringJoin["(/ (", ToString[indicesDers[[indname, indindx + 1]]],
      "+(k-1)*NDRTP_P, k=1,NSP_P) /)"],
     {indname, indindx + 1}
    ];,
   {k, Length[indlist]}
  ];
  , (* ELSE *)
 ];

 indicesEqns = indicesEqns // padPointers;
 indicesVars = indicesVars // padPointers;
 indicesDers = indicesDers // padPointers;

];
```

◇ *Code symbols*

```
Options[writeCodeSymbols] =
  {
   codeform → FortranForm,
   indent → "      ",
   tocode → {}
  };

writeCodeSymbols::usage =
  "writeCodeSymbols[symbolarray,output,options___]:\n
Writes the symbols (variables, derivatives, etc.} in a given
    PDE in code form. Options work as in Jacobian components routines.";
```

```
writeCodeSymbols[symbolarray_, output_, options___] :=

  Module[

   {
    lhs, rhs, eqn, sublistlength,
    sublistwidth, symbol, symbol1, symbol2, symbol3, symbolinchars,
    paddedlength = 4, (** number of chars/spaces between 1st char and " =" **)
    MyCodeForm = codeform /. {options} /. Options[writeCodeSymbols],
    MyCodeIndent = indent /. {options} /. Options[writeCodeSymbols],
    MyToCode = tocode /. {options} /. Options[writeCodeSymbols]
   },

   sublistwidth = Dimensions[symbolarray][[1]];
   If[sublistwidth == 1,
    symbol = ToString[symbolarray[[1]]];
    If[output == 1 || output == 3, Print[symbol, " = ", "\n"],];
    symbolinchars = Characters[symbol];
    If[Length[symbolinchars] < paddedlength,
     symbol = StringJoin[PadRight[symbolinchars, paddedlength, " "]],];
    lhs = StringJoin[MyCodeIndent, symbol, " = "];
    MyToCode = Flatten[Append[MyToCode, {"\n", lhs}]];
    , (** ELSE **)
    symbol1 = ToString[symbolarray[[1]]];
    symbol2 = ToString[symbolarray[[2]]];
    symbol3 = ToString[symbolarray[[3]]];
    If[output == 1 || output == 3,
     Print[symbol1, " = ", symbol2, "(", symbol3, ",ind)", "\n"],];
    symbolinchars = Characters[symbol1];
    If[Length[symbolinchars] < paddedlength,
     symbol1 = StringJoin[PadRight[symbolinchars, paddedlength, " "]],];
    lhs = StringJoin[MyCodeIndent, symbol1, " = "];
    rhs = StringJoin[symbol2, "(", symbol3, ",ind)"];
    MyToCode = Flatten[Append[MyToCode, {"\n", lhs, rhs}]];
   ];

   If[FreeQ[{options}, tocode],
    If[(output == 2 || output == 3), Print @@ MyToCode,],
    Return[MyToCode]
   ];

  ];
```

◇ *Equation residuals*

```
Options[writeEquationResidual] =
  {
   codeform → FortranAssign,
   indent → "      ",
   tocode → {}
  };

writeEquationResidual::usage =
  "writeEquationResidual[pde,pdeindex,output,options___]:\n
Writes the equation residual for a given PDE
    in code form. Options work as in Jacobian components routines.";

writeEquationResidual[pdein_, pdeindex_, output_, options___] :=

  Module[

   {
    lhs, rhs, eqn,
    MyCodeForm = codeform /. {options} /. Options[writeEquationResidual],
    MyCodeIndent = indent /. {options} /. Options[writeCodeSymbols],
    MyToCode = tocode /. {options}  /. Options[writeEquationResidual]
   },

   pde = pdein;

   If[output == 1 || output == 3, Print["F =  ", pde, "\n"],];
   lhs = StringJoin[MyCodeIndent, "EQ0(", indicesEqns[[pdeindex, 1]], ",ind) = "];
   rhs = MyCodeForm[pde,
      AssignBreak → {1000, "\n     &"},
      AssignIndent → "",
      AssignOptimize → False,
      AssignPrecision → Infinity,
      AssignTemporary → {"tmp0", Sequence}
     ][[1]] ;
   MyToCode = Flatten[Append[MyToCode, {"\n", lhs, rhs}]];

   If[FreeQ[{options}, tocode],
    If[(output == 2 || output == 3), Print@@MyToCode,],
    Return[MyToCode]
   ];

  ];
```

◇ *Local components of the Jacobian*

```
Options[jacobianLocalComponents] =
  {
   codeform → FortranAssign,
   indent → "      ",
   tocode → {}
  };

jacobianLocalComponents::usage =
  "jacobianLocalComponents[pde,pdeindex,vars,output,options___]:\n
* 'pde' is the equation in symbolic form; '
    pdeindex' is the number of this PDE in the system of PDEs\n
* 'vars' is a list of all possible variables (unknowns)
    which can occur in the equations\n
* 'output' is a flag which determines whether Mathematica summaries of
    the components or actual code is written (1 = Mathematica only,
    2 = code only, 3 = both); current settings generate Fortran code\n
* 'options' is a sequence of rules (not required)";

jacobianLocalComponents[pdein_, pdeindex_, vars_, output_, options___] :=

  Module[

   {
    lhs, rhs,
    MyCodeForm = codeform /. {options} /. Options[jacobianLocalComponents],
    MyCodeIndent = indent /. {options} /. Options[writeCodeSymbols],
    MyToCode = tocode /. {options} /. Options[jacobianLocalComponents]
   },

   pde = pdein //. jacobianIdealGasRule;

   Do[

    var = vars[[k]];
    tmp0 = D[pde, var];
    If[var === T, tmp1 = LACPt D[pde, LACP], tmp1 = 0];
    If[var === T, tmp2 = CPt D[pde, CP], tmp2 = 0];
    If[var === T, tmp3 = LACPtt D[pde, LACPt], tmp3 = 0];
    If[var === T, tmp4 = CPtt D[pde, CPt], tmp4 = 0];
    If[var === YK, tmp5 = WMy D[pde, WM], tmp5 = 0];
    DJ[pdeindex, k] =
     ( (tmp0 + tmp1 + tmp2 + tmp3 + tmp4 + tmp5) // Expand) //. jacobianIdealGasRestoreRules;

    If[
     DJ[pdeindex, k] =!= 0,
     If[output == 1 || output == 3,
      Print["d F", ToString[pdeindex],
       " / d (", vars[[k]], ") =  ", DJ[pdeindex, k], "\n"],];
```

```
    lhs = StringJoin[MyCodeIndent, "DJ(", indicesEqns[[pdeindex, l]],
      ",", indicesVars[[k, l]], ",ind) = "];
    rhs = MyCodeForm[DJ[pdeindex, k],
       AssignBreak → {1000, "\n     &"},
       AssignIndent → "",
       AssignOptimize → False,
       AssignPrecision → Infinity,
       AssignTemporary → {"tmp0", Sequence}
      ][[1]] ;
    MyToCode = Flatten[Append[MyToCode, {"\n", lhs, rhs}]];
    , (* ELSE *)
   ];

   , {k, Length[vars]}

  ];

  If[FreeQ[{options}, tocode],
   If[(output == 2 || output == 3), Print @@ MyToCode,],
   Return[MyToCode]
  ];

 ];
```

◇ *Time derivative (diagonal) terms of the Jacobian*

```
Options[jacobianTimeDerivatives] =
  {
   codeform → FortranAssign,
   indent → "      ",
   tocode → {}
  };

jacobianTimeDerivatives::usage =
  "jacobianTimeDerivatives[pde,pdeindex,vars,output,options___]:\n
* 'pde' is the equation in symbolic form; '
    pdeindex' is the number of this PDE in the system of PDEs\n
* 'vars' is a list of all possible variables (unknowns)
    which can occur in the equations\n
* 'output' is a flag which determines whether Mathematica summaries of
    the components or actual code is written (1 = Mathematica only,
    2 = code only, 3 = both); current settings generate Fortran code\n
* 'options' is a sequence of rules (not required)";

jacobianTimeDerivatives[pdein_, pdeindex_, vars_, output_, options___] :=

  Module[

   {
    lhs, rhs1, rhs2,
    MyCodeForm = codeform /. {options} /. Options[jacobianLocalComponents],
```

```
 MyCodeIndent = indent /. {options} /. Options[writeCodeSymbols],
 MyToCode = tocode /. {options}  /. Options[jacobianLocalComponents],
 tders, tmptders
},

tmptders = Outer[ StringJoin, Map[ToString, vars], {"t"} ] // Flatten;
tders = Map[ToExpression, tmptders];
Remove[tmptders];

pde = pdein //. jacobianIdealGasRule;

Do[

 tmp = D[pde, tders[[k]]];
 DJT[pdeindex, k] = tmp //. jacobianIdealGasRestoreRules;

 If[
  DJT[pdeindex, k] =!= 0,
  If[output == 1 || output == 3,
   Print["d F", ToString[pdeindex],
    " / d (", tders[[k]], ") =  ", DJT[pdeindex, k], "\n"],];
  lhs = StringJoin[MyCodeIndent, "DJ(", indicesEqns[[pdeindex, 1]],
    ",", indicesVars[[k, 1]], ",ind) = "];
  rhs1 = StringJoin["DJ(", indicesEqns[[pdeindex, 1]], ",",
    indicesVars[[k, 1]], ",ind) + "];
  rhs2 = MyCodeForm[DJT[pdeindex, k] ddt,
     AssignBreak → {1000, "\n      &"},
     AssignIndent → "",
     AssignOptimize → False,
     AssignPrecision → Infinity,
     AssignTemporary → {"tmp0", Sequence}
    ][[1]] ;
  MyToCode = Flatten[Append[MyToCode, {"\n", lhs, rhs1, rhs2}]];
  , (* ELSE *)
 ];

 , {k, Length[vars]}

];

If[FreeQ[{options}, tocode],
 If[(output == 2 || output == 3), Print @@ MyToCode,],
 Return[MyToCode]
];

];
```

◇ *Spatial components of the Jacobian*

```
Options[jacobianSpatialComponents] =
  {
   codeform → FortranAssign,
   indent → "      ",
   tocode → {}
  };

jacobianSpatialComponents::usage =
  "jacobianSpatialComponents[pde,pdeindex,derivs,output,options___]:\n
* 'pde' is the equation in symbolic form; '
    pdeindex' is the number of this PDE in the system of PDEs\n
* 'derivs' is a list of all possible derivatives which can occur in the equations\n
* 'output' is a flag which determines whether Mathematica summaries
    of the components or actual code is written (1 = Mathematica only,
    2 = code only, 3 = both); current settings generate Fortran code\n
* 'options' is a sequence of rules (not required)";
```

```
jacobianSpatialComponents[pdein_, pdeindex_, derivs_, output_, options___] :=

 Module[

  {
   lhs, rhs, eqn,
   MyCodeForm = codeform /. {options} /. Options[jacobianSpatialComponents],
   MyCodeIndent = indent /. {options} /. Options[writeCodeSymbols],
   MyToCode = tocode /. {options}  /. Options[jacobianSpatialComponents]
  },

  pde = pdein //. jacobianIdealGasRule;

  Do[

   tmp = D[pde, derivs[[k]]];
   DQ[pdeindex, k] = tmp //. jacobianIdealGasRestoreRules;

   If[
    DQ[pdeindex, k] =!= 0,
    If[output == 1 || output == 3,
     Print["d F", ToString[pdeindex],
      " / d (", derivs[[k]], ") =  ", DQ[pdeindex, k], "\n"],];
    lhs = StringJoin[MyCodeIndent, "DQ(", indicesEqns[[pdeindex, 1]],
      ",", indicesDers[[k, 1]], ",ind) = "];
    rhs = MyCodeForm[DQ[pdeindex, k],
       AssignBreak → {1000, "\n      &"},
       AssignIndent → "",
       AssignOptimize → False,
       AssignPrecision → Infinity,
       AssignTemporary → {"tmp0", Sequence}
      ][[1]] ;
    MyToCode = Flatten[Append[MyToCode, {"\n", lhs, rhs}]];
    , (* ELSE *)
   ];

   , {k, Length[derivs]}

  ];

  If[FreeQ[{options}, tocode],
   If[(output == 2 || output == 3), Print @@ MyToCode,],
   Return[MyToCode]
  ];

 ];
```

RESIDUALS & JACOBIAN - INTERIOR POINTS

■ Pressure

◇ *Residuals*

```
pde = ContinuityEquation;
pde = pde /. assumptionsRules // Expand;
pde = (r * pde) // Expand;
PDE[1] = toCodeNotation[pde] //. equationsIdealGasRules
Remove[pde];
```

$$\frac{Pt\ r\ RHO}{P} - \frac{r\ RHO\ Tt}{T} + RHO\ U + \frac{Pr\ r\ RHO\ U}{P} - \frac{r\ RHO\ Tr\ U}{T} + r\ RHO\ Ur + \frac{Pz\ r\ RHO\ W}{P} - \frac{r\ RHO\ Tz\ W}{T} + r\ RHO\ Wz + \frac{r\ RHO\ U\ WM\ YKr}{WK} + \frac{r\ RHO\ WM\ YKt}{WK} + \frac{r\ RHO\ W\ WM\ YKz}{WK}$$

◇ *Jacobian*

```
jacobianLocalComponents[PDE[1], 1, variables, 1];
jacobianTimeDerivatives[PDE[1], 1, variables, 1];
jacobianSpatialComponents[PDE[1], 1, derivatives, 1];
```

$$d\ F1\ /\ d\ (P) = -\frac{r\ RHO\ Tt}{P\ T} + \frac{RHO\ U}{P} - \frac{r\ RHO\ Tr\ U}{P\ T} + \frac{r\ RHO\ Ur}{P} - \frac{r\ RHO\ Tz\ W}{P\ T} + \frac{r\ RHO\ Wz}{P} + \frac{r\ RHO\ U\ WM\ YKr}{P\ WK} + \frac{r\ RHO\ WM\ YKt}{P\ WK} + \frac{r\ RHO\ W\ WM\ YKz}{P\ WK}$$

$$d\ F1\ /\ d\ (U) = RHO + \frac{Pr\ r\ RHO}{P} - \frac{r\ RHO\ Tr}{T} + \frac{r\ RHO\ WM\ YKr}{WK}$$

$$d\ F1\ /\ d\ (W) = \frac{Pz\ r\ RHO}{P} - \frac{r\ RHO\ Tz}{T} + \frac{r\ RHO\ WM\ YKz}{WK}$$

$$d\ F1\ /\ d\ (T) = -\frac{Pt\ r\ RHO}{P\ T} + \frac{2\ r\ RHO\ Tt}{T^2} - \frac{RHO\ U}{T} - \frac{Pr\ r\ RHO\ U}{P\ T} + \frac{2\ r\ RHO\ Tr\ U}{T^2} - \frac{r\ RHO\ Ur}{T} - \frac{Pz\ r\ RHO\ W}{P\ T} + \frac{2\ r\ RHO\ Tz\ W}{T^2} - \frac{r\ RHO\ Wz}{T} - \frac{r\ RHO\ U\ WM\ YKr}{T\ WK} - \frac{r\ RHO\ WM\ YKt}{T\ WK} - \frac{r\ RHO\ W\ WM\ YKz}{T\ WK}$$

$$d\ F1\ /\ d\ (YK) = -\frac{Pt\ r\ RHO\ WM}{P\ WK} + \frac{r\ RHO\ Tt\ WM}{T\ WK} - \frac{RHO\ U\ WM}{WK} - \frac{Pr\ r\ RHO\ U\ WM}{P\ WK} + \frac{r\ RHO\ Tr\ U\ WM}{T\ WK} - \frac{r\ RHO\ Ur\ WM}{WK} - \frac{Pz\ r\ RHO\ W\ WM}{P\ WK} + \frac{r\ RHO\ Tz\ W\ WM}{T\ WK} - \frac{r\ RHO\ WM\ Wz}{WK} - \frac{2\ r\ RHO\ U\ WM^2\ YKr}{WK^2} - \frac{2\ r\ RHO\ WM^2\ YKt}{WK^2} - \frac{2\ r\ RHO\ W\ WM^2\ YKz}{WK^2}$$

$$d\ F1\ /\ d\ (Pt) = \frac{r\ RHO}{P}$$

$$d\ F1\ /\ d\ (Tt) = -\frac{r\ RHO}{T}$$

$$d\,F1\,/\,d\,(YKt) = \frac{r\,RHO\,WM}{WK}$$

$$d\,F1\,/\,d\,(Pr) = \frac{r\,RHO\,U}{P}$$

$$d\,F1\,/\,d\,(Pz) = \frac{r\,RHO\,W}{P}$$

$$d\,F1\,/\,d\,(Ur) = r\,RHO$$

$$d\,F1\,/\,d\,(Wz) = r\,RHO$$

$$d\,F1\,/\,d\,(Tr) = -\frac{r\,RHO\,U}{T}$$

$$d\,F1\,/\,d\,(Tz) = -\frac{r\,RHO\,W}{T}$$

$$d\,F1\,/\,d\,(YKr) = \frac{r\,RHO\,U\,WM}{WK}$$

$$d\,F1\,/\,d\,(YKz) = \frac{r\,RHO\,W\,WM}{WK}$$

■ Radial velocity

◇ *Residuals*

```
pde = NavierStokesEquation[[1]];
pde = pde /. assumptionsRules // FullSimplify // Expand;
pde = (r^2 * pde) // Expand;
PDE[2] = toCodeNotation[pde] //. equationsIdealGasRules
Remove[pde];
```

$$Pr\,r^2 + \frac{4\,LACP\,PR\,U}{3} + \frac{2}{3}\,LACPt\,PR\,r\,Tr\,U - \frac{4}{3}\,LACP\,PR\,r\,Ur - \frac{4}{3}\,LACPt\,PR\,r^2\,Tr\,Ur + r^2\,RHO\,U\,Ur - \frac{4}{3}\,LACP\,PR\,r^2\,Urr + r^2\,RHO\,Ut - LACPt\,PR\,r^2\,Tz\,Uz - LACP\,PR\,r^2\,Uzz + r^2\,RHO\,Uz\,W - LACPt\,PR\,r^2\,Tz\,Wr - \frac{1}{3}\,LACP\,PR\,r^2\,Wrz + \frac{2}{3}\,LACPt\,PR\,r^2\,Tr\,Wz$$

◇ *Jacobian*

```
jacobianLocalComponents[PDE[2], 2, variables, 1];
jacobianTimeDerivatives[PDE[2], 2, variables, 1];
jacobianSpatialComponents[PDE[2], 2, derivatives, 1];
```

$$d\,F2\,/\,d\,(P) = \frac{r^2\,RHO\,U\,Ur}{P} + \frac{r^2\,RHO\,Ut}{P} + \frac{r^2\,RHO\,Uz\,W}{P}$$

$$d\,F2\,/\,d\,(U) = \frac{4\,LACP\,PR}{3} + \frac{2}{3}\,LACPt\,PR\,r\,Tr + r^2\,RHO\,Ur$$

$$d\,F2\,/\,d\,(W) = r^2\,RHO\,Uz$$

$$d\,F2\,/\,d\,(T) = \frac{4\,LACPt\,PR\,U}{3} + \frac{2}{3}\,LACPtt\,PR\,r\,Tr\,U - \frac{4}{3}\,LACPt\,PR\,r\,Ur - \frac{4}{3}\,LACPtt\,PR\,r^2\,Tr\,Ur - \frac{r^2\,RHO\,U\,Ur}{T} - \frac{4}{3}\,LACPt\,PR\,r^2\,Urr - \frac{r^2\,RHO\,Ut}{T} - LACPtt\,PR\,r^2\,Tz\,Uz - LACPt\,PR\,r^2\,Uzz - \frac{r^2\,RHO\,Uz\,W}{T} - LACPtt\,PR\,r^2\,Tz\,Wr - \frac{1}{3}\,LACPt\,PR\,r^2\,Wrz + \frac{2}{3}\,LACPtt\,PR\,r^2\,Tr\,Wz$$

$$d\,F2\,/\,d\,(YK) = -\frac{r^2\,RHO\,U\,Ur\,WM}{WK} - \frac{r^2\,RHO\,Ut\,WM}{WK} - \frac{r^2\,RHO\,Uz\,W\,WM}{WK}$$

$$d\,F2\,/\,d\,(Ut) = r^2\,RHO$$

$$d\,F2\,/\,d\,(Pr) = r^2$$

$$d\,F2\,/\,d\,(Ur) = -\frac{4}{3}\,LACP\,PR\,r - \frac{4}{3}\,LACPt\,PR\,r^2\,Tr + r^2\,RHO\,U$$

$$d\,F2\,/\,d\,(Uz) = -LACPt\,PR\,r^2\,Tz + r^2\,RHO\,W$$

$$d\,F2\,/\,d\,(Urr) = -\frac{4}{3}\,LACP\,PR\,r^2$$

$$d\,F2\,/\,d\,(Uzz) = -LACP\,PR\,r^2$$

$$d\,F2\,/\,d\,(Wr) = -LACPt\,PR\,r^2\,Tz$$

$$d\,F2\,/\,d\,(Wz) = \frac{2}{3}\,LACPt\,PR\,r^2\,Tr$$

$$d\,F2\,/\,d\,(Wrz) = -\frac{1}{3}\,LACP\,PR\,r^2$$

$$d\,F2\,/\,d\,(Tr) = \frac{2}{3}\,LACPt\,PR\,r\,U - \frac{4}{3}\,LACPt\,PR\,r^2\,Ur + \frac{2}{3}\,LACPt\,PR\,r^2\,Wz$$

$$d\,F2\,/\,d\,(Tz) = -LACPt\,PR\,r^2\,Uz - LACPt\,PR\,r^2\,Wr$$

■ Axial velocity

◇ *Residuals*

```
pde = NavierStokesEquation[[3]];
pde = pde /. assumptionsRules // FullSimplify // Expand;
pde = (r * pde) // Expand;
PDE[3] = toCodeNotation[pde] //. equationsIdealGasRules
Remove[pde];
```

$$\text{Pz r} - \text{GZ r RHO} + \frac{2}{3}\,\text{LACPt PR Tz U} + \frac{2}{3}\,\text{LACPt PR r Tz Ur} - \frac{1}{3}\,\text{LACP PR r Urz} - \frac{\text{LACP PR Uz}}{3} - \text{LACPt PR r Tr Uz} - \text{LACP PR Wr} - \text{LACPt PR r Tr Wr} + \text{r RHO U Wr} - \text{LACP PR r Wrr} + \text{r RHO Wt} - \frac{4}{3}\,\text{LACPt PR r Tz Wz} + \text{r RHO W Wz} - \frac{4}{3}\,\text{LACP PR r Wzz}$$

◇ *Jacobian*

```
jacobianLocalComponents[PDE[3], 3, variables, 1];
jacobianTimeDerivatives[PDE[3], 3, variables, 1];
jacobianSpatialComponents[PDE[3], 3, derivatives, 1];
```

$$d\,F3\,/\,d\,(P) = -\frac{GZ\,r\,RHO}{P} + \frac{r\,RHO\,U\,Wr}{P} + \frac{r\,RHO\,Wt}{P} + \frac{r\,RHO\,W\,Wz}{P}$$

$$d\,F3\,/\,d\,(U) = \frac{2\,LACPt\,PR\,Tz}{3} + r\,RHO\,Wr$$

$$d\,F3\,/\,d\,(W) = r\,RHO\,Wz$$

$$d\,F3\,/\,d\,(T) = \frac{GZ\,r\,RHO}{T} + \frac{2}{3}\,LACPtt\,PR\,Tz\,U + \frac{2}{3}\,LACPtt\,PR\,r\,Tz\,Ur - \frac{1}{3}\,LACPt\,PR\,r\,Urz - \frac{LACPt\,PR\,Uz}{3} - LACPtt\,PR\,r\,Tr\,Uz - LACPt\,PR\,Wr - LACPtt\,PR\,r\,Tr\,Wr - \frac{r\,RHO\,U\,Wr}{T} - LACPt\,PR\,r\,Wrr - \frac{r\,RHO\,Wt}{T} - \frac{4}{3}\,LACPtt\,PR\,r\,Tz\,Wz - \frac{r\,RHO\,W\,Wz}{T} - \frac{4}{3}\,LACPt\,PR\,r\,Wzz$$

$$d\,F3\,/\,d\,(YK) = \frac{GZ\,r\,RHO\,WM}{WK} - \frac{r\,RHO\,U\,WM\,Wr}{WK} - \frac{r\,RHO\,WM\,Wt}{WK} - \frac{r\,RHO\,W\,WM\,Wz}{WK}$$

$$d\,F3\,/\,d\,(Wt) = r\,RHO$$

$$d\,F3\,/\,d\,(Pz) = r$$

$$d\,F3\,/\,d\,(Ur) = \frac{2}{3}\,LACPt\,PR\,r\,Tz$$

$$d\,F3\,/\,d\,(Uz) = -\frac{LACP\,PR}{3} - LACPt\,PR\,r\,Tr$$

$$d\,F3\,/\,d\,(Urz) = -\frac{1}{3}\,LACP\,PR\,r$$

$$d\,F3\,/\,d\,(Wr) = -LACP\,PR - LACPt\,PR\,r\,Tr + r\,RHO\,U$$

$$d\,F3\,/\,d\,(Wz) = -\frac{4}{3}\,LACPt\,PR\,r\,Tz + r\,RHO\,W$$

$$d\,F3\,/\,d\,(Wrr) = -LACP\,PR\,r$$

$$d\,F3\,/\,d\,(Wzz) = -\frac{4}{3}\,LACP\,PR\,r$$

$$d\,F3\,/\,d\,(Tr) = -LACPt\,PR\,r\,Uz - LACPt\,PR\,r\,Wr$$

$$d\,F3\,/\,d\,(Tz) = \frac{2\,LACPt\,PR\,U}{3} + \frac{2}{3}\,LACPt\,PR\,r\,Ur - \frac{4}{3}\,LACPt\,PR\,r\,Wz$$

■ Temperature

◇ *Residuals*

```
pde = EnergyEquation;
pde = pde /. assumptionsRules // Expand;
pde = (r * pde) // Expand;
PDE[4] = toCodeNotation[pde] //. equationsIdealGasRules
Remove[pde];
```

$$-\frac{\text{HEAT r}}{\text{CP}} - \text{LACP Tr} - \frac{\text{CPt LACP r Tr}^2}{\text{CP}} - \text{LACPt r Tr}^2 - \text{LACP r Trr} +$$
$$\text{r RHO Tt} - \frac{\text{CPt LACP r Tz}^2}{\text{CP}} - \text{LACPt r Tz}^2 - \text{LACP r Tzz} + \text{r RHO Tr U} + \text{r RHO Tz W}$$

◇ *Jacobian*

```
jacobianLocalComponents[PDE[4], 4, variables, 1];
jacobianTimeDerivatives[PDE[4], 4, variables, 1];
jacobianSpatialComponents[PDE[4], 4, derivatives, 1];
```

$$d\,F4\,/\,d\,(P) = \frac{r\,RHO\,Tt}{P} + \frac{r\,RHO\,Tr\,U}{P} + \frac{r\,RHO\,Tz\,W}{P}$$

$$d\,F4\,/\,d\,(U) = r\,RHO\,Tr$$

$$d\,F4\,/\,d\,(W) = r\,RHO\,Tz$$

$$d\,F4\,/\,d\,(T) = \frac{CPt\,HEAT\,r}{CP^2} - LACPt\,Tr + \frac{CPt^2\,LACP\,r\,Tr^2}{CP^2} - \frac{CPtt\,LACP\,r\,Tr^2}{CP} - \frac{CPt\,LACPt\,r\,Tr^2}{CP} - LACPtt\,r\,Tr^2 - LACPt\,r\,Trr - \frac{r\,RHO\,Tt}{T} + \frac{CPt^2\,LACP\,r\,Tz^2}{CP^2} - \frac{CPtt\,LACP\,r\,Tz^2}{CP} - \frac{CPt\,LACPt\,r\,Tz^2}{CP} - LACPtt\,r\,Tz^2 - LACPt\,r\,Tzz - \frac{r\,RHO\,Tr\,U}{T} - \frac{r\,RHO\,Tz\,W}{T}$$

$$d\,F4\,/\,d\,(YK) = -\frac{r\,RHO\,Tt\,WM}{WK} - \frac{r\,RHO\,Tr\,U\,WM}{WK} - \frac{r\,RHO\,Tz\,W\,WM}{WK}$$

$$d\,F4\,/\,d\,(Tt) = r\,RHO$$

$$d\,F4\,/\,d\,(Tr) = -LACP - \frac{2\,CPt\,LACP\,r\,Tr}{CP} - 2\,LACPt\,r\,Tr + r\,RHO\,U$$

$$d\,F4\,/\,d\,(Tz) = -\frac{2\,CPt\,LACP\,r\,Tz}{CP} - 2\,LACPt\,r\,Tz + r\,RHO\,W$$

$$d\,F4\,/\,d\,(Trr) = -LACP\,r$$

$$d\,F4\,/\,d\,(Tzz) = -LACP\,r$$

■ Species

◇ *Residuals*

```
pde = SpeciesConservation;
pde = pde /. assumptionsRules // Expand;
pde = (r * pde) // Expand;
PDE[5] = toCodeNotation[pde] //. equationsIdealGasRules
Remove[pde];
```

$$-r\,SOURCE + RHO\,UC\,YK + \frac{Pr\,r\,RHQ\,UC\,YK}{P} - \frac{r\,RHO\,Tr\,UC\,YK}{T} + r\,RHO\,UCr\,YK + \frac{Pz\,r\,RHO\,WC\,YK}{P} - \frac{r\,RHO\,Tz\,WC\,YK}{T} + r\,RHO\,WCz\,YK - \frac{LACP\,YKr}{LEk} - \frac{LACPt\,r\,Tr\,YKr}{LEk} + r\,RHO\,U\,YKr + r\,RHO\,UC\,YKr + \frac{r\,RHO\,UC\,WM\,YK\,YKr}{WK} - \frac{LACP\,r\,YKrr}{LEk} + r\,RHO\,YKt - \frac{LACPt\,r\,Tz\,YKz}{LEk} + r\,RHO\,W\,YKz + r\,RHO\,WC\,YKz + \frac{r\,RHO\,WC\,WM\,YK\,YKz}{WK} - \frac{LACP\,r\,YKzz}{LEk}$$

◇ *Jacobian*

```
jacobianLocalComponents[PDE[5], 5, variables, 1];
jacobianTimeDerivatives[PDE[5], 5, variables, 1];
jacobianSpatialComponents[PDE[5], 5, derivatives, 1];
```

$$d\,F5\,/\,d\,(P) = \frac{RHO\,UC\,YK}{P} - \frac{r\,RHO\,Tr\,UC\,YK}{P\,T} + \frac{r\,RHO\,UCr\,YK}{P} - \frac{r\,RHO\,Tz\,WC\,YK}{P\,T} + \frac{r\,RHO\,WCz\,YK}{P} + \frac{r\,RHO\,U\,YKr}{P} + \frac{r\,RHO\,UC\,YKr}{P} + \frac{r\,RHO\,UC\,WM\,YK\,YKr}{P\,WK} + \frac{r\,RHO\,YKt}{P} + \frac{r\,RHO\,W\,YKz}{P} + \frac{r\,RHO\,WC\,YKz}{P} + \frac{r\,RHO\,WC\,WM\,YK\,YKz}{P\,WK}$$

$$d\,F5\,/\,d\,(U) = r\,RHO\,YKr$$

$$d\,F5\,/\,d\,(W) = r\,RHO\,YKz$$

$$d\,F5\,/\,d\,(T) = -\frac{RHO\,UC\,YK}{T} - \frac{Pr\,r\,RHO\,UC\,YK}{P\,T} + \frac{2\,r\,RHO\,Tr\,UC\,YK}{T^2} - \frac{r\,RHO\,UCr\,YK}{T} - \frac{Pz\,r\,RHO\,WC\,YK}{P\,T} + \frac{2\,r\,RHO\,Tz\,WC\,YK}{T^2} - \frac{r\,RHO\,WCz\,YK}{T} - \frac{LACPt\,YKr}{LEk} - \frac{LACPtt\,r\,Tr\,YKr}{LEk} - \frac{r\,RHO\,U\,YKr}{T} - \frac{r\,RHO\,UC\,YKr}{T} - \frac{r\,RHO\,UC\,WM\,YK\,YKr}{T\,WK} - \frac{LACPt\,r\,YKrr}{LEk} - \frac{r\,RHO\,YKt}{T} - \frac{LACPtt\,r\,Tz\,YKz}{LEk} - \frac{r\,RHO\,W\,YKz}{T} - \frac{r\,RHO\,WC\,YKz}{T} - \frac{r\,RHO\,WC\,WM\,YK\,YKz}{T\,WK} - \frac{LACPt\,r\,YKzz}{LEk}$$

$$d\,F5\,/\,d\,(YK) = RHO\,UC + \frac{Pr\,r\,RHO\,UC}{P} - \frac{r\,RHO\,Tr\,UC}{T} + r\,RHO\,UCr + \frac{Pz\,r\,RHO\,WC}{P} - \frac{r\,RHO\,Tz\,WC}{T} + r\,RHO\,WCz - \frac{RHO\,UC\,WM\,YK}{WK} - \frac{Pr\,r\,RHO\,UC\,WM\,YK}{P\,WK} + \frac{r\,RHO\,Tr\,UC\,WM\,YK}{T\,WK} - \frac{r\,RHO\,UCr\,WM\,YK}{WK} - \frac{Pz\,r\,RHO\,WC\,WM\,YK}{P\,WK} + \frac{r\,RHO\,Tz\,WC\,WM\,YK}{T\,WK} - \frac{r\,RHO\,WCz\,WM\,YK}{WK} - \frac{r\,RHO\,U\,WM\,YKr}{WK} - \frac{2\,r\,RHO\,UC\,WM^2\,YK\,YKr}{WK^2} - \frac{r\,RHO\,WM\,YKt}{WK} - \frac{r\,RHO\,W\,WM\,YKz}{WK} - \frac{2\,r\,RHO\,WC\,WM^2\,YK\,YKz}{WK^2}$$

$$d\,F5\,/\,d\,(YKt) = r\,RHO$$

$$d\,F5\,/\,d\,(Pr) = \frac{r\,RHO\,UC\,YK}{P}$$

$$d\,F5\,/\,d\,(Pz) = \frac{r\,RHO\,WC\,YK}{P}$$

$$d\,F5\,/\,d\,(Tr) = -\frac{r\,RHO\,UC\,YK}{T} - \frac{LACPt\,r\,YKr}{LEk}$$

$$d\,F5\,/\,d\,(Tz) = -\frac{r\,RHO\,WC\,YK}{T} - \frac{LACPt\,r\,YKz}{LEk}$$

$$d\,F5\,/\,d\,(YKr) = -\frac{LACP}{LEk} - \frac{LACPt\,r\,Tr}{LEk} + r\,RHO\,U + r\,RHO\,UC + \frac{r\,RHO\,UC\,WM\,YK}{WK}$$

$$d\,F5\,/\,d\,(YKz) = -\frac{LACPt\,r\,Tz}{LEk} + r\,RHO\,W + r\,RHO\,WC + \frac{r\,RHO\,WC\,WM\,YK}{WK}$$

$$d\,F5\,/\,d\,(YKrr) = -\frac{LACP\,r}{LEk}$$

$$d\,F5\,/\,d\,(YKzz) = -\frac{LACP\,r}{LEk}$$

RESIDUALS & JACOBIAN - INLET BOUNDARY

RESIDUALS & JACOBIAN - AXIS OF SYMMETRY

RESIDUALS & JACOBIAN - WALL BOUNDARY

RESIDUALS & JACOBIAN - OUTLET BOUNDARY

Output code material for *point.f90*

Output code material for *orj.f*

APPENDIX II:

Fortran Code for
RESIDUALS and JACOBIAN Subroutines
of the Implicit-Compact Solver

(partial listing of a code created using the
software tool presented in Appendix I)

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!%%                                                                          %%
!%%                                                                          %%
!%%                              RESIDUALS                                   %%
!%%                                                                          %%
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      SUBROUTINE RESIDUALS (ISTEP)

      USE DimensioningParameters
      USE ProblemParameters
      USE MethodParameters, ONLY : iDSPACE,iDTIME,                         &
     &                             TFLOW,DT0,DT1,DT2,                      &
     &                             KVFIX, NFIX
      USE Pointers
      USE DiscretizedProblem ,   ONLY : NVAR, NSP, NR, NZ, NODES, NEL,     &
     &                                  RI, ZJ,                            &
     &                                  SF, S, S1, S2, SX,                 &
     &                                  SD,                                &
     &                                  EQ0, EQ1, CT, F, SB1,              &
     &                                  VELinlet
      USE Differentiation
      USE OneStepChemistry_Methane
      USE SimplifiedTransport
      USE Time_IO_Debug,         ONLY : WTRES

      IMPLICIT DOUBLE PRECISION (A-H,O-Z)

      double precision dSdt(NVAR_P,NODES_P)

      double precision YK(NSP_P), YKt(NSP_P), YKinlet(NSP_P)
      double precision YKr(NSP_P), YKz(NSP_P), YKrr(NSP_P), YKzz(NSP_P)

      double precision LACP, LACPt, LEK(NSP_P), DK(NSP_P), DKt(NSP_P),     &
! stoichiometric coeffs (# mols produced/destroyed)
     &                 NUK(NSP_P),                                         &
! mass coeffs: NUK(k)*WK(k)
     &                 NUKWK(NSP_P),                                       &
     &                 WK(NSP_P), SOURCE(NSP_P)

      save isavsd  ! should be initialized to zero by compiler
      save igetrsf ! should be initialized to zero by compiler
      save ioutflag

!.....upwinding.....
      double precision CONVRDER(NVAR_P,NODES_P),CONVZDER(NVAR_P,NODES_P)

      double precision CONVECT(NVAR_P)

!.....upwinding.....
      iUPWIND = 1    ! 1 = upwind; 2 = 2nd order centered (convective terms)

      timephase = 2*PI*TFLOW/Period
```

```
!-------------------------------------------------------------------------------
!     INITIALIZATION
!-------------------------------------------------------------------------------

      call DZERO(F,NEL)
      call DZERO(EQ0,NEL)
      call DZERO(dSdt,NEL)
      call DZERO(CT,NEL)

!.....upwinding.....
      call DZERO(CONVRDER,NEL)
      call DZERO(CONVZDER,NEL)
      call DZERO(CONVECT,NVAR)

      if (iSTEP.eq.0) then
         call DZERO(EQ1,NEL)
      endif

!-------------------------------------------------------------------------------
!     SPATIAL DERIVATIVES
!-------------------------------------------------------------------------------

      if (iDSPACE.eq.1) then
         call SPATIAL_LO (S)
      else
         call SPATIAL_CS (S)
      endif

!-------------------------------------------------------------------------------
!     RESIDUAL - INTERIOR NODES
!-------------------------------------------------------------------------------

      DO j = 2, NZ-1
      DO i = 2, NR-1

         ind = (j-1)*NR + i

         r   = RI(i)

         P0  = AtmPressure
         P2  = S(MvarP,ind)
!         P   = P0 + P2
         P   = P0

         U    = S(MvarU ,ind)
         W    = S(MvarW ,ind)
         T    = S(MvarT ,ind)
         YK   = S(MvarYK,ind)

         Pr   = SD(KderPr  ,ind)
         Pz   = SD(KderPz  ,ind)
         Ur   = SD(KderUr  ,ind)
         Uz   = SD(KderUz  ,ind)
         Urr  = SD(KderUrr ,ind)
         Uzz  = SD(KderUzz ,ind)
         Urz  = SD(KderUrz ,ind)
         Wr   = SD(KderWr  ,ind)
```

```
Wz   = SD(KderWz  ,ind)
Wrr  = SD(KderWrr ,ind)
Wzz  = SD(KderWzz ,ind)
Wrz  = SD(KderWrz ,ind)
Tr   = SD(KderTr  ,ind)
Tz   = SD(KderTz  ,ind)
Trr  = SD(KderTrr ,ind)
Tzz  = SD(KderTzz ,ind)
YKr  = SD(KderYKr ,ind)
YKz  = SD(KderYKz ,ind)
YKrr = SD(KderYKrr,ind)
YKzz = SD(KderYKzz,ind)

Pt  = 0.d0 ! set to zero here when computing EQ0
Ut  = 0.d0 ! set to zero here when computing EQ0
Wt  = 0.d0 ! set to zero here when computing EQ0
Tt  = 0.d0 ! set to zero here when computing EQ0
YKt = 0.d0 ! set to zero here when computing EQ0

WK   = MolecularWeights
WM   = MixtureWeight (YK, WK)
RHO  = P*WM/(RU*T)
PDYN = P2
GZ   = GRAV

CP    = SpecificHeat(T)
CPt   = 0.d0            ! SpecificHeat_ddT(T)
LACP  = LambdaCp(T)     ! lambda/Cp
LACPt = LambdaCp_ddT(T) ! d(lambda/Cp)/dT
PRN   = PrandtlNumber
LEK   = LewisNumbers

DK    = LACP/(LEK*RHO)
DKt   = LACPt/(LEK*RHO)+LACP/(LEK*RHO*T)
UC    = dot_product(DK,YKr) ! correction vel. for mass conservation
UCr   = dot_product(DK,YKrr)+dot_product((DKt*Tr),YKr)
WC    = dot_product(DK,YKz) ! correction vel. for mass conservation
WCz   = dot_product(DK,YKzz)+dot_product((DKt*Tz),YKz)

NUK     = SignedStoichiometricCoeffs
NUKWK   = NUK*WK
! (nu_O * Wk_O)/(nu_F * Wk_F) @ stoichiometric conditions
sstoich = NUKWK(2)/NUKWK(1)
Yfuel   = YK(1)
Yox     = YK(2)
Yfuel_F = 1.d0    ! mass fraction of fuel in the Fuel Stream
Yox_A   = 0.232d0 ! mass fraction of oxydizer in the Air Stream
Z       = Z_mixfrac(sstoich, Yfuel, Yox, Yfuel_F, Yox_A)
PHI     = PHI_equivratio (Z, Yfuel_F, Yox_A)
omega   = MolarProductionRate(PHI, RHO, YK, Wk, T)
SOURCE  = (NUKWK/abs(NUK(1))) * omega
q       = HeatReleasePerMole(PHI)
HEAT    = q * omega

!       density gradients, time derivative

RHOrP = (Pr*r*RHO*U)/P
```

```
      RHOrT = -(r*RHO*Tr*U)/T
      RHOrY =  sum( ((r*RHO*U*WM*YKr)/WK) )

      RHOr  = RHOrP+RHOrT+RHOrY

      RHOzP = (Pz*r*RHO*W)/P
      RHOzT = -(r*RHO*Tz*W)/T
      RHOzY = sum( ((r*RHO*W*WM*YKz)/WK) )

      RHOz  = RHOzP+RHOzT+RHOzY

      RHOt  = 0.d0 ! leave off time terms till later...

!     convective terms

      CONVECT(MeqnP)  =     (U*RHOr + W*RHOz) * r
      CONVECT(MeqnU)  = RHO*(U*Ur   + W*Uz)   * r**2
      CONVECT(MeqnW)  = RHO*(U*Wr   + W*Wz)   * r
      CONVECT(MeqnT)  = RHO*(U*Tr   + W*Tz)   * r
      CONVECT(MeqnYK) = RHO*(U*YKr  + W*YKz)  * r

      if (iUPWIND.eq.1 .and. iDSPACE.eq.1) then
         call UPWIND(ind,i,j,r,RHO,CONVRDER,CONVZDER,CONVECT)\
      endif


!     steady part of the residuals

      EQ0(MeqnP ,ind) = r*RHOt+RHO*U+r*RHO*Ur+r*RHO*Wz+CONVECT(MeqnP)

      EQ0(MeqnU ,ind) = (4*LACP*PRN*U)/3.d0+(2*LACPt*PRN*r*Tr*U)/3.d0      &
     &                  -(4*LACP*PRN*r*Ur)/3.d0+Pr*(r*r)                  &
     &                  -(4*LACPt*PRN*Tr*Ur*(r*r))/3.d0                   &
     &                  -(4*LACP*PRN*Urr*(r*r))/3.d0+RHO*Ut*(r*r)         &
     &                  -LACPt*PRN*Tz*Uz*(r*r)-LACP*PRN*Uzz*(r*r)         &
     &                  -LACPt*PRN*Tz*Wr*(r*r)                            &
     &                  -(LACP*PRN*Wrz*(r*r))/3.d0                        &
     &                  +(2*LACPt*PRN*Tr*Wz*(r*r))/3.d0                   &
     &                  +CONVECT(MeqnU)

      EQ0(MeqnW ,ind) = Pz*r-GZ*r*RHO+(2*LACPt*PRN*Tz*U)/3.d0             &
     &                  +(2*LACPt*PRN*r*Tz*Ur)/3.d0-(LACP*PRN*r*Urz)/3.d0 &
     &                  -(LACP*PRN*Uz)/3.d0-LACPt*PRN*r*Tr*Uz-LACP*PRN*Wr &
     &                  -LACPt*PRN*r*Tr*Wr-LACP*PRN*r*Wrr                 &
     &                  +r*RHO*Wt-(4*LACPt*PRN*r*Tz*Wz)/3.d0              &
     &                  -(4*LACP*PRN*r*Wzz)/3.d0                          &
     &                  +CONVECT(MeqnW)

      EQ0(MeqnT ,ind) = -((HEAT*r)/CP)-LACP*Tr-LACP*r*Trr+r*RHO*Tt        &
     &                  -LACP*r*Tzz                                       &
     &                  -(CPt*LACP*r*(Tr*Tr))/CP-LACPt*r*(Tr*Tr)          &
     &                  -(CPt*LACP*r*(Tz*Tz))/CP-LACPt*r*(Tz*Tz)          &
     &                  +CONVECT(MeqnT)

      EQ0(MeqnYK,ind) = -(r*SOURCE)+RHO*UC*YK+r*RHOr*UC*YK+r*RHO*UCr*YK   &
     &                  +r*RHOz*WC*YK+r*RHO*WCz*YK-(LACP*YKr)/LEK         &
     &                  -(LACPt*r*Tr*YKr)/LEK+r*RHO*UC*YKr                &
```

```
     &                        -(LACP*r*YKrr)/LEK+r*RHO*YKt-(LACPt*r*Tz*YKz)/LEK  &
     &                        +r*RHO*WC*YKz-(LACP*r*YKzz)/LEK                    &
     &                        +CONVECT(MeqnYK)

      ENDDO
      ENDDO

!-------------------------------------------------------------------------------
!     RESIDUAL - BOUNDARY NODES
!-------------------------------------------------------------------------------

!     CONDITIONS AT THE INLET: BOUNDARY 1

      j = 1

      DO i = 2, NR

         ind = (j-1)*NR + i

         r   = RI(i)

         P0  = AtmPressure
         P2  = S(MvarP,ind)
!         P   = P0 + P2
         P   = P0

         U    = S(MvarU  ,ind)
         W    = S(MvarW  ,ind)
         T    = S(MvarT  ,ind)
         YK   = S(MvarYK ,ind)

         Pz  = SD(KderPz  ,ind)
         Ur  = SD(KderUr  ,ind)
         Uz  = SD(KderUz  ,ind)
         Urz = SD(KderUrz ,ind)
         Wr  = SD(KderWr  ,ind)
         Wz  = SD(KderWz  ,ind)
         Wrr = SD(KderWrr ,ind)
         Wzz = SD(KderWzz ,ind)
         Tr  = SD(KderTr  ,ind)
         Tz  = SD(KderTz  ,ind)

         Wt   = 0.d0
         if (iDTIME.gt.1)                                                   &
     &      Wt = VELinlet(i)*MODULATION*2*PI/PERIOD*cos(timephase)

         WK   = MolecularWeights
         WM   = MixtureWeight (YK, WK)
         RHO  = P*WM/(RU*T)
         PDYN = P2
         GZ   = GRAV

         LACP  = LambdaCp(T)      ! lambda/Cp
         LACPt = LambdaCp_ddT(T) ! d(lambda/Cp)/dT
         PRN   = PrandtlNumber

         Winlet  = SB1(MvarW ,i)
```

```
          Tinlet  = SB1(MvarT ,i)
          YKinlet = SB1(MvarYK,i)

          EQ0(MeqnP ,ind) = Pz*r-GZ*r*RHO+(2*LACPt*PRN*Tz*U)/3.d0                &
     &                       +(2*LACPt*PRN*r*Tz*Ur)/3.d0-(LACP*PRN*r*Urz)/3.d0  &
     &                       -(LACP*PRN*Uz)/3.d0-LACPt*PRN*r*Tr*Uz-LACP*PRN*Wr  &
     &                       -LACPt*PRN*r*Tr*Wr+r*RHO*U*Wr-LACP*PRN*r*Wrr       &
     &                       +r*RHO*Wt-(4*LACPt*PRN*r*Tz*Wz)/3.d0+r*RHO*W*Wz    &
     &                       -(4*LACP*PRN*r*Wzz)/3.d0
          EQ0(MeqnU ,ind) = U
          EQ0(MeqnW ,ind) = W-Winlet
          EQ0(MeqnT ,ind) = T-Tinlet
          EQ0(MeqnYK,ind) = YK-YKinlet

        ENDDO

!.....Special treatment (for pressure BC) at symmetry-inlet corner pt
        i   = 1
        ind = 1
        U    = S(MvarU  ,ind)
        W    = S(MvarW  ,ind)
        T    = S(MvarT  ,ind)
        YK   = S(MvarYK ,ind)
        Pr   = SD(KderPr  ,ind)
        Winlet  = SB1(MvarW ,i)
        Tinlet  = SB1(MvarT ,i)
        YKinlet = SB1(MvarYK,i)
        EQ0(MeqnP ,ind) = Pr
        EQ0(MeqnU ,ind) = U
        EQ0(MeqnW ,ind) = W-Winlet
        EQ0(MeqnT ,ind) = T-Tinlet
        EQ0(MeqnYK,ind) = YK-YKinlet


                          <<<<< ... CUT ... >>>>>



!=============================================================================
!     TIME-DERIVATIVES
!=============================================================================

!-------------
!     iDTIME=0 : STEADY STATE - NO TIME DERIVATIVE TERMS
!-------------

        IF (iDTIME.eq.0) THEN

           do ind = 1,nodes
           do keq = 1,nvar
              F(keq,ind) = EQ0(keq,ind)
           enddo
           enddo

           goto 999
```

```
!-------------
!     iDTIME=1 : BACKWARD EULER DISCRETIZATION IN PSEUDO-TIME
!-------------

      ELSEIF (iDTIME.eq.1) THEN

!        INTERIOR POINTS

         do j=2,NZ-1
         do i=2,NR-1

            ind = (j-1)*NR+i

            r   = RI(i)

            P0  = AtmPressure
            P2  = S(MvarP,ind)
!            P   = P0 + P2
            P   = P0

            T   = S(MvarT  ,ind)
            YK  = S(MvarYK ,ind)

            WK  = MolecularWeights
            WM  = MixtureWeight (YK, WK)
            RHO = P*WM/(RU*T)

            CT(2,ind) = RHO * r**2
            CT(3,ind) = RHO * r
            CT(4,ind) = RHO * r
            do k = 1,NSP
               CT(4+k,ind) = RHO * r
            enddo

            Pt    = (S(MvarP ,ind)-S1(MvarP ,ind))/DT0
            Tt    = (S(MvarT ,ind)-S1(MvarT ,ind))/DT0
            YKt   = (S(MvarYK,ind)-S1(MvarYK,ind))/DT0
            RHOtP =  (Pt*r*RHO)/P
            RHOtT = -(r*RHO*Tt)/T
            RHOtY = sum( ((r*RHO*WM*YKt)/WK) )
            RHOt  = RHOtP ! +RHOtT+RHOtY
            F(MvarP,ind) = RHOt + EQ0(MvarP,ind)
            do k = 2,NVAR
               dSdt(k,ind) = (S(k,ind)-S1(k,ind))/DT0
               F(k,ind) = CT(k,ind) * dSdt(k,ind) + EQ0(k,ind)
            enddo

         enddo
         enddo


                        <<<<< ... CUT ... >>>>>
```

```
!-------------------------------------------------------------------------------
!     SOME VARIABLES MAY BE FIXED
!-------------------------------------------------------------------------------

999   continue

      if (KVFIX.gt.0) then
         do ind=1,NODES
            do k=1,NVAR
               if (NFIX(k).eq.1) then
                  F(k,ind)=S(k,ind)-SX(k,ind)
                  EQ0(k,ind) = 0.d0
               endif
            enddo
         enddo
      endif

!     monitor norms of steady and unsteady residuals

      call NORM2(NEL,EQ0,eq0nrm)
      call NORM2(NEL,F,  fnrm)
      dnel=sqrt(dble(NEL))
      fnrm=fnrm/dnel
      eq0nrm=eq0nrm/dnel

      if (ISTEP.ne.-1) then
         write( 6,*) '@@@ eq0nrm = ',eq0nrm
         write( 6,*) '@@@   fnrm = ',  fnrm
         write(16,*) '@@@ eq0nrm = ',eq0nrm
         write(16,*) '@@@   fnrm = ',  fnrm
      endif

      RETURN
      END
```

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
!%%                                                                          %%
!%%                              JACOBIAN                                    %%
!%%                                                                          %%
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

      SUBROUTINE JACOBIAN

      USE DimensioningParameters
      USE ProblemParameters
      USE MethodParameters, ONLY : iDSPACE,iDTIME,                          &
     &                             TFLOW,DT0,DT1,DT2,                       &
     &                             iMATVEC,                                 &
     &                             KVFIX, NFIX
      USE Pointers
      USE DiscretizedProblem ,   ONLY : NVAR, NSP, NR, NZ, NODES, NEL,      &
     &                                  NDRTP,                              &
     &                                  RI, ZJ,                             &
     &                                  SF, S, S1, S2, SX,                  &
     &                                  SD,                                 &
     &                                  DJ, DQ, CTDJ, F,                    &
     &                                  VELinlet
      USE Differentiation
      USE OneStepChemistry_Methane
      USE SimplifiedTransport
      USE Time_IO_Debug,         ONLY : CPUREST, SECONDM

      IMPLICIT DOUBLE PRECISION (A-H,O-Z)

      double precision YK(NSP_P), YKt(NSP_P), YKinlet(NSP_P)
      double precision YKr(NSP_P), YKz(NSP_P), YKrr(NSP_P), YKzz(NSP_P)

      double precision LACP, LACPt, LACPtt, LEK(NSP_P), DK(NSP_P), DKt(NSP_P), &
! stoichiometric coeffs (# mols produced/destroyed)
     &                 NUK(NSP_P),                                          &
! mass coeffs: NUK(k)*WK(k)
     &                 NUKWK(NSP_P),                                        &
     &                 WK(NSP_P), SOURCE(NSP_P), SOURCEp(NSP_P),            &
     &                 SOURCEt(NSP_P), SOURCEy(NSP_P,NSP_P), HEATy(NSP_P),  &
     &                 omegay(NSP_P)

      double precision DSPERT(NODES_P), SHOLD(NVAR_P,NODES_P)
      double precision F0(NVAR_P,NODES_P)

      iNumericalDJ = 1    ! 1 - numerical, 0 - analytical

      timephase = 2*PI*TFLOW/Period

!-------------------------------------------------------------------------------
!     INITIALIZE
!-------------------------------------------------------------------------------

      NDJ = NVAR_P*NVAR_P*NODES_P  ! size of array DJ
      NDQ = NVAR_P*LDQ_P*NODES_P   ! size of array DQ

      call DZERO(DJ,NDJ)
      call DZERO(DJN,NDJ) ! numerical DJ
```

```
      call DZERO(DQ,NDQ)
      call DZERO(CTDJ,NEL)

      if (iDTIME.eq.0) then
         ! NO TIME DERIVATIVES: steady-state solve
      elseif (iDTIME.eq.1) then
         ddt=1.D0/DT0       ! pseudo-time Implicit Euler
      elseif (iDTIME.eq.2) then
         ddt=1.D0/DT0       ! time-dependent Implicit Euler
      elseif (iDTIME.eq.3) then
         ddt=1.D0/DT0       ! time-dependent Crank-Nicolson
      elseif (iDTIME.eq.4) then
         ddt=3.D0/(2.d0*DT0)! time-dependent 2nd order BDF
      else
         write(*,*) 'STOP: other time discrets. not yet implemented'
         stop
      endif

!-----------------------------------------------------------------------------
!     NUMERICAL DJ
!-----------------------------------------------------------------------------

      IF (iNumericalDJ.eq.1) THEN

         call RESIDUALS (-1)

         F0 = F     ! save initial residual
         SHOLD = S  ! save current solution S

         DO kvar = 1,NVAR

            DO j = 2, NZ-1
            DO i = 2, NR-1
               ind = (j-1)*NR+i
               tmp = S(kvar,ind)
               DSPERT(ind) = tmp*1.d-8 + 1.d-8
               S(kvar,ind) = tmp + DSPERT(ind)
            ENDDO
            ENDDO

            call RESIDUALS (-1)    ! compute perturbed residual

            DO j = 2, NZ-1
            DO i = 2, NR-1
            ind = (j-1)*NR+i
            DO keqn = 1,NVAR
               DJ(keqn,kvar,ind)=(F(keqn,ind)-F0(keqn,ind))/DSPERT(ind)
            ENDDO
            ENDDO
            ENDDO

            S = SHOLD

         ENDDO
         call RESIDUALS (-1)

      ENDIF
```

```
!-------------------------------------------------------------------------------
!     SPATIAL DERIVATIVES
!-------------------------------------------------------------------------------

      if (iDSPACE.eq.1) then  ! LO method
         call SPATIAL_LO (S)
      else
         call SPATIAL_CS (S)
      endif


!-------------------------------------------------------------------------------
!     JACOBIAN - INTERIOR NODES
!-------------------------------------------------------------------------------

      DO j = 2, NZ-1
      DO i = 2, NR-1

         ind = (j-1)*NR + i

         r   = RI(i)

         P0  = AtmPressure
         P2  = S(MvarP,ind)
!         P   = P0 + P2
         P   = P0

         U    = S(MvarU  ,ind)
         W    = S(MvarW  ,ind)
         T    = S(MvarT  ,ind)
         YK   = S(MvarYK ,ind)

         Pr  = SD(KderPr  ,ind)
         Pz  = SD(KderPz  ,ind)
         Ur  = SD(KderUr  ,ind)
         Uz  = SD(KderUz  ,ind)
         Urr = SD(KderUrr ,ind)
         Uzz = SD(KderUzz ,ind)
         Urz = SD(KderUrz ,ind)
         Wr  = SD(KderWr  ,ind)
         Wz  = SD(KderWz  ,ind)
         Wrr = SD(KderWrr ,ind)
         Wzz = SD(KderWzz ,ind)
         Wrz = SD(KderWrz ,ind)
         Tr  = SD(KderTr  ,ind)
         Tz  = SD(KderTz  ,ind)
         Trr = SD(KderTrr ,ind)
         Tzz = SD(KderTzz ,ind)
         YKr = SD(KderYKr ,ind)
         YKz = SD(KderYKz ,ind)
         YKrr= SD(KderYKrr,ind)
         YKzz= SD(KderYKzz,ind)

!        time derivatives
         Pt  = 0.d0
         Ut  = 0.d0
```

```
      Wt  = 0.d0
      Tt  = 0.d0
      YKt = 0.d0
      IF (iDTIME.ge.1 .and. iDTIME.le.3) THEN        ! ps-time IE, IE, CN
         Pt  = ddt*(S(MvarP ,ind)-S1(MvarP ,ind))
         Ut  = ddt*(S(MvarU ,ind)-S1(MvarU ,ind))
         Wt  = ddt*(S(MvarW ,ind)-S1(MvarW ,ind))
         Tt  = ddt*(S(MvarT ,ind)-S1(MvarT ,ind))
         YKt = ddt*(S(MvarYK,ind)-S1(MvarYK,ind))
      ELSEIF (iDTIME.eq.4) THEN                      ! BDF2
         Pt  = 0.5d0/DT0*(3.d0*S(MvarP ,ind)-4.d0*S1(MvarP ,ind)           &
     &              +1.d0*S2(MvarP ,ind))
         Ut  = 0.5d0/DT0*(3.d0*S(MvarU ,ind)-4.d0*S1(MvarU ,ind)           &
     &              +1.d0*S2(MvarU  ,ind))
         Wt  = 0.5d0/DT0*(3.d0*S(MvarW ,ind)-4.d0*S1(MvarW ,ind)           &
     &              +1.d0*S2(MvarW  ,ind))
         Tt  = 0.5d0/DT0*(3.d0*S(MvarT ,ind)-4.d0*S1(MvarT ,ind)           &
     &              +1.d0*S2(MvarT  ,ind))
         YKt = 0.5d0/DT0*(3.d0*S(MvarYK,ind)-4.d0*S1(MvarYK,ind)           &
     &              +1.d0*S2(MvarYK ,ind))
      ENDIF

      WK   = MolecularWeights
      WM   = MixtureWeight (YK, WK)
      RHO  = P*WM/(RU*T)
      PDYN = P2
      GZ   = GRAV

      CP     = SpecificHeat(T)
      CPt    = 0.d0              ! SpecificHeat_ddT(T)
      CPtt   = 0.d0              ! SpecificHeat_d2dT2(T)
      LACP   = LambdaCp(T)       ! lambda/Cp
      LACPt  = LambdaCp_ddT(T)   ! d(lambda/Cp)/dT
      LACPtt = LambdaCp_d2dT2(T) ! d2(lambda/Cp)/dT2
      PRN    = PrandtlNumber
      LEK    = LewisNumbers

      DK    = LACP/(LEK*RHO)
      DKt   = LACPt/(LEK*RHO)+LACP/(LEK*RHO*T)
      UC    = dot_product(DK,YKr)  ! correction vel. for mass conservation
      UCr   = dot_product(DK,YKrr)+dot_product((DKt*Tr),YKr)
      WC    = dot_product(DK,YKz)  ! correction vel. for mass conservation
      WCz   = dot_product(DK,YKzz)+dot_product((DKt*Tz),YKz)

      NUK     = SignedStoichiometricCoeffs
      NUKWK   = NUK*WK
      ! (nu_O * Wk_O)/(nu_F * Wk_F) @ stoichiometric conditions
      sstoich = NUKWK(2)/NUKWK(1)
      Yfuel   = YK(1)
      Yox     = YK(2)
      Yfuel_F = 1.d0    ! mass fraction of fuel in the Fuel Stream
      Yox_A   = 0.232d0 ! mass fraction of oxydizer in the Air Stream
      Z       = Z_mixfrac(sstoich, Yfuel, Yox, Yfuel_F, Yox_A)
      PHI     = PHI_equivratio (Z, Yfuel_F, Yox_A)
      omega   = MolarProductionRate(PHI, RHO, YK, Wk, T)
      omegap  = MolarProductionRate_ddP(PHI, RHO, YK, Wk, T)
      omegat  = MolarProductionRate_ddT(PHI, RHO, YK, Wk, T)
```

```
      omegay  = MolarProductionRate_ddYK(PHI, RHO, YK, Wk, T)
      SOURCE  = (NUKWK/abs(NUK(1))) * omega
      SOURCEp = (NUKWK/abs(NUK(1))) * omegap
      SOURCEt = (NUKWK/abs(NUK(1))) * omegat
      ! outer product of NUWK * (omegay)^T
      SOURCEy = spread((NUKWK/abs(NUK(1))),2,NSP_P )*spread(omegay,1,NSP_P)
      q       = HeatReleasePerMole(PHI)
      HEAT    = q * omega
      HEATp   = q * omegap
      HEATt   = q * omegat
      HEATy   = q * omegay

!     LOCAL COMPONENTS - MAIN BLOCK DIAGONAL

      IF (iNumericalDJ.ne.1) THEN

!.....Analytical DJ

      DJ(MeqnP ,MvarP ,ind) = -((r*RHO*Tt)/(P*T))+(RHO*U)/P                  &
     &                        -(r*RHO*Tr*U)/(P*T)+(r*RHO*Ur)/P               &
     &                        -(r*RHO*Tz*W)/(P*T)+(r*RHO*Wz)/P               &
     &                        +sum( ((r*RHO*U*WM*YKr)/(P*WK)) )              &
     &                        +sum( ((r*RHO*WM*YKt)/(P*WK)) )                &
     &                        +sum( ((r*RHO*W*WM*YKz)/(P*WK)) )
      DJ(MeqnP ,MvarU ,ind) = RHO+(Pr*r*RHO)/P-(r*RHO*Tr)/T                  &
     &                        +sum( ((r*RHO*WM*YKr)/WK) )
      DJ(MeqnP ,MvarW ,ind) = (Pz*r*RHO)/P-(r*RHO*Tz)/T                      &
     &                        +sum( ((r*RHO*WM*YKz)/WK) )
      DJ(MeqnP ,MvarT ,ind) = -((Pt*r*RHO)/(P*T))+(2*r*RHO*Tt)/T**2          &
     &                        -(RHO*U)/T-(Pr*r*RHO*U)/(P*T)                  &
     &                        +(2*r*RHO*Tr*U)/T**2-(r*RHO*Ur)/T              &
     &                        -(Pz*r*RHO*W)/(P*T)+(2*r*RHO*Tz*W)/T**2        &
     &                        -(r*RHO*Wz)/T                                  &
     &                        -sum( ((r*RHO*U*WM*YKr)/(T*WK)) )              &
     &                        -sum( ((r*RHO*WM*YKt)/(T*WK)) )                &
     &                        -sum( ((r*RHO*W*WM*YKz)/(T*WK)) )
      DJ(MeqnP ,MvarYK,ind) = -((Pt*r*RHO*WM)/(P*WK))+(r*RHO*Tt*WM)/(T*WK)   &
     &                        -(RHO*U*WM)/WK-(Pr*r*RHO*U*WM)/(P*WK)          &
     &                        +(r*RHO*Tr*U*WM)/(T*WK)-(r*RHO*Ur*WM)/WK       &
     &                        -(Pz*r*RHO*W*WM)/(P*WK)                        &
     &                        +(r*RHO*Tz*W*WM)/(T*WK)-(r*RHO*WM*Wz)/WK       &
     &                        -(2*r*RHO*U*YKr*(WM*WM))/WK**2                 &
     &                        -(2*r*RHO*YKt*(WM*WM))/WK**2                   &
     &                        -(2*r*RHO*W*YKz*(WM*WM))/WK**2

      DJ(MeqnU ,MvarP ,ind) = (RHO*U*Ur*(r*r))/P+(RHO*Ut*(r*r))/P            &
     &                        +(RHO*Uz*W*(r*r))/P
      DJ(MeqnU ,MvarU ,ind) = (4*LACP*PRN)/3.d0+(2*LACPt*PRN*r*Tr)/3.d0      &
     &                        +RHO*Ur*(r*r)
      DJ(MeqnU ,MvarW ,ind) = RHO*Uz*(r*r)
      DJ(MeqnU ,MvarT ,ind) = (4*LACPt*PRN*U)/3.d0                           &
     &                        +(2*LACPtt*PRN*r*Tr*U)/3.d0                    &
     &                        -(4*LACPt*PRN*r*Ur)/3.d0                       &
     &                        -(4*LACPtt*PRN*Tr*Ur*(r*r))/3.d0               &
     &                        -(RHO*U*Ur*(r*r))/T                            &
     &                        -(4*LACPt*PRN*Urr*(r*r))/3.d0                  &
     &                        -(RHO*Ut*(r*r))/T-LACPtt*PRN*Tz*Uz*(r*r)       &
```

```
&                              -LACPt*PRN*Uzz*(r*r)-(RHO*Uz*W*(r*r))/T       &
&                              -LACPtt*PRN*Tz*Wr*(r*r)                       &
&                              -(LACPt*PRN*Wrz*(r*r))/3.d0                   &
&                              +(2*LACPtt*PRN*Tr*Wz*(r*r))/3.d0
      DJ(MeqnU ,MvarYK,ind) = -((RHO*U*Ur*WM*(r*r))/WK)                      &
&                              -(RHO*Ut*WM*(r*r))/WK-(RHO*Uz*W*WM*(r*r))/WK

      DJ(MeqnW ,MvarP ,ind) = -((GZ*r*RHO)/P)+(r*RHO*U*Wr)/P+(r*RHO*Wt)/P   &
&                              +(r*RHO*W*Wz)/P
      DJ(MeqnW ,MvarU ,ind) = (2*LACPt*PRN*Tz)/3.d0+r*RHO*Wr
      DJ(MeqnW ,MvarW ,ind) = r*RHO*Wz
      DJ(MeqnW ,MvarT ,ind) = (GZ*r*RHO)/T+(2*LACPtt*PRN*Tz*U)/3.d0          &
&                              +(2*LACPtt*PRN*r*Tz*Ur)/3.d0                  &
&                              -(LACPt*PRN*r*Urz)/3.d0-(LACPt*PRN*Uz)/3.d0   &
&                              -LACPtt*PRN*r*Tr*Uz-LACPt*PRN*Wr              &
&                              -LACPtt*PRN*r*Tr*Wr-(r*RHO*U*Wr)/T            &
&                              -LACPt*PRN*r*Wrr-(r*RHO*Wt)/T                 &
&                              -(4*LACPtt*PRN*r*Tz*Wz)/3.d0-(r*RHO*W*Wz)/T   &
&                              -(4*LACPt*PRN*r*Wzz)/3.d0
      DJ(MeqnW ,MvarYK,ind) = (GZ*r*RHO*WM)/WK-(r*RHO*U*WM*Wr)/WK            &
&                              -(r*RHO*WM*Wt)/WK-(r*RHO*W*WM*Wz)/WK

      DJ(MeqnT ,MvarP ,ind) = (r*RHO*Tt)/P+(r*RHO*Tr*U)/P+(r*RHO*Tz*W)/P
      DJ(MeqnT ,MvarU ,ind) = r*RHO*Tr
      DJ(MeqnT ,MvarW ,ind) = r*RHO*Tz
      DJ(MeqnT ,MvarT ,ind) = (CPt*HEAT*r)/CP**2-LACPt*Tr-LACPt*r*Trr        &
&                              -(r*RHO*Tt)/T-LACPt*r*Tzz-(r*RHO*Tr*U)/T      &
&                              -(r*RHO*Tz*W)/T-(CPtt*LACP*r*(Tr*Tr))/CP      &
&                              -(CPt*LACPt*r*(Tr*Tr))/CP-LACPtt*r*(Tr*Tr)    &
&                              +(LACP*r*(CPt*CPt)*(Tr*Tr))/CP**2             &
&                              -(CPtt*LACP*r*(Tz*Tz))/CP                     &
&                              -(CPt*LACPt*r*(Tz*Tz))/CP-LACPtt*r*(Tz*Tz)    &
&                              +(LACP*r*(CPt*CPt)*(Tz*Tz))/CP**2+HEATt
      DJ(MeqnT ,MvarYK,ind) = -((r*RHO*Tt*WM)/WK)-(r*RHO*Tr*U*WM)/WK         &
&                              -(r*RHO*Tz*W*WM)/WK+HEATy

      DJ(MeqnYK,MvarP ,ind) = (RHO*UC*YK)/P-(r*RHO*Tr*UC*YK)/(P*T)           &
&                              +(r*RHO*UCr*YK)/P-(r*RHO*Tz*WC*YK)/(P*T)      &
&                              +(r*RHO*WCz*YK)/P+(r*RHO*U*YKr)/P             &
&                              +(r*RHO*UC*YKr)/P                             &
&                              +(r*RHO*UC*WM*YK*YKr)/(P*WK)+(r*RHO*YKt)/P    &
&                              +(r*RHO*W*YKz)/P+(r*RHO*WC*YKz)/P             &
&                              +(r*RHO*WC*WM*YK*YKz)/(P*WK)
      DJ(MeqnYK,MvarU ,ind) = r*RHO*YKr
      DJ(MeqnYK,MvarW ,ind) = r*RHO*YKz
      DJ(MeqnYK,MvarT ,ind) = -((RHO*UC*YK)/T)-(Pr*r*RHO*UC*YK)/(P*T)        &
&                              +(2*r*RHO*Tr*UC*YK)/T**2-(r*RHO*UCr*YK)/T     &
&                              -(Pz*r*RHO*WC*YK)/(P*T)                       &
&                              +(2*r*RHO*Tz*WC*YK)/T**2-(r*RHO*WCz*YK)/T     &
&                              -(LACPt*YKr)/LEK-(LACPtt*r*Tr*YKr)/LEK        &
&                              -(r*RHO*U*YKr)/T-(r*RHO*UC*YKr)/T             &
&                              -(r*RHO*UC*WM*YK*YKr)/(T*WK)                  &
&                              -(LACPt*r*YKrr)/LEK-(r*RHO*YKt)/T             &
&                              -(LACPtt*r*Tz*YKz)/LEK-(r*RHO*W*YKz)/T        &
&                              -(r*RHO*WC*YKz)/T                             &
&                              -(r*RHO*WC*WM*YK*YKz)/(T*WK)                  &
&                              -(LACPt*r*YKzz)/LEK
```

```
        FORALL (k=1:NSP)
! roughly correct: dF(YK)/dYK is full since F(YK) contains RHO, a fn of all YK
        DJ(MeqnYK(k),MvarYK(k),ind) = RHO*UC+(Pr*r*RHO*UC)/P-(r*RHO*Tr*UC)/T  &
     &                                +r*RHO*UCr+(Pz*r*RHO*WC)/P              &
     &                                -(r*RHO*Tz*WC)/T+r*RHO*WCz              &
     &                                -(RHO*UC*WM*YK(k))/WK(k)                &
     &                                -(Pr*r*RHO*UC*WM*YK(k))/(P*WK(k))       &
     &                                +(r*RHO*Tr*UC*WM*YK(k))/(T*WK(k))       &
     &                                -(r*RHO*UCr*WM*YK(k))/WK(k)             &
     &                                -(Pz*r*RHO*WC*WM*YK(k))/(P*WK(k))       &
     &                                +(r*RHO*Tz*WC*WM*YK(k))/(T*WK(k))       &
     &                                -(r*RHO*WCz*WM*YK(k))/WK(k)             &
     &                                -(r*RHO*U*WM*YKr(k))/WK(k)              &
     &                                -(r*RHO*WM*YKt(k))/WK(k)                &
     &                                -(r*RHO*W*WM*YKz(k))/WK(k)              &
     &                                -(2*r*RHO*UC*YK(k)*YKr(k)               &
     &                                          *(WM*WM))/WK(k)**2            &
     &                                -(2*r*RHO*WC*YK(k)*YKz(k)               &
     &                                          *(WM*WM))/WK(k)**2
        END FORALL
        DJ(MeqnYK ,MvarYK ,ind) = DJ(MeqnYK ,MvarYK ,ind) + SOURCEy
        ! N.B. SOURCEy is a matrix

        IF (iDTIME.eq.3) THEN  ! Crank-Nicolson only

           DO k2 = 1, NVAR
           DO k1 = 1, NVAR
              DJ(k1,k2,ind) = 0.5d0*DJ(k1,k2,ind)
           ENDDO
           ENDDO

        ENDIF

!       ADD TIME DERIVATIVE TERMS TO LOCAL COMPONENTS (DIAGONAL ONLY)

        IF (iDTIME.ne.0) THEN  ! omit time derivatives for steady problem

           DJ(MeqnP ,MvarP ,ind) = DJ(MeqnP ,MvarP ,ind) + (ddt*r*RHO)/P
           IF (iDTIME.ne.1) THEN
           DJ(MeqnP ,MvarT ,ind) = DJ(MeqnP ,MvarT ,ind) -((ddt*r*RHO)/T)
           DJ(MeqnP ,MvarYK,ind) = DJ(MeqnP ,MvarYK,ind) + (ddt*r*RHO*WM)/WK
           ENDIF
           DJ(MeqnU ,MvarU ,ind) = DJ(MeqnU  ,MvarU  ,ind) + ddt*RHO*(r*r)
           DJ(MeqnW ,MvarW ,ind) = DJ(MeqnW  ,MvarW  ,ind) + ddt*r*RHO
           DJ(MeqnT ,MvarT ,ind) = DJ(MeqnT  ,MvarT  ,ind) + ddt*r*RHO
           FORALL (k=1:NSP)
           DJ(MeqnYK(k),MvarYK(k),ind) = DJ(MeqnYK(k),MvarYK(k),ind)          &
     &                                     + ddt*r*RHO
           END FORALL

        ENDIF

!.......END Analytical DJ

        ENDIF

!       SPATIAL COMPONENTS
```

```
      DQ(MeqnP ,KderPr  ,ind) = (r*RHO*U)/P
      DQ(MeqnP ,KderPz  ,ind) = (r*RHO*W)/P
      DQ(MeqnP ,KderUr  ,ind) = r*RHO
      DQ(MeqnP ,KderWz  ,ind) = r*RHO
      DQ(MeqnP ,KderTr  ,ind) = -((r*RHO*U)/T)
      DQ(MeqnP ,KderTz  ,ind) = -((r*RHO*W)/T)
      DQ(MeqnP ,KderYKr ,ind) = (r*RHO*U*WM)/WK
      DQ(MeqnP ,KderYKz ,ind) = (r*RHO*W*WM)/WK

      DQ(MeqnU ,KderPr  ,ind) = r*r
      DQ(MeqnU ,KderUr  ,ind) = (-4*LACP*PRN*r)/3.d0                          &
     &                          -(4*LACPt*PRN*Tr*(r*r))/3.d0+RHO*U*(r*r)
      DQ(MeqnU ,KderUz  ,ind) = -(LACPt*PRN*Tz*(r*r))+RHO*W*(r*r)
      DQ(MeqnU ,KderUrr ,ind) = (-4*LACP*PRN*(r*r))/3.d0
      DQ(MeqnU ,KderUzz ,ind) = -(LACP*PRN*(r*r))
      DQ(MeqnU ,KderWr  ,ind) = -(LACPt*PRN*Tz*(r*r))
      DQ(MeqnU ,KderWz  ,ind) = (2*LACPt*PRN*Tr*(r*r))/3.d0
      DQ(MeqnU ,KderWrz ,ind) = -(LACP*PRN*(r*r))/3.d0
      DQ(MeqnU ,KderTr  ,ind) = (2*LACPt*PRN*r*U)/3.d0                        &
     &                          -(4*LACPt*PRN*Ur*(r*r))/3.d0                  &
     &                          +(2*LACPt*PRN*Wz*(r*r))/3.d0
      DQ(MeqnU ,KderTz  ,ind) = -(LACPt*PRN*Uz*(r*r))-LACPt*PRN*Wr*(r*r)

      DQ(MeqnW ,KderPz  ,ind) = r
      DQ(MeqnW ,KderUr  ,ind) = (2*LACPt*PRN*r*Tz)/3.d0
      DQ(MeqnW ,KderUz  ,ind) = -(LACP*PRN)/3.d0-LACPt*PRN*r*Tr
      DQ(MeqnW ,KderUrz ,ind) = -(LACP*PRN*r)/3.d0
      DQ(MeqnW ,KderWr  ,ind) = -(LACP*PRN)-LACPt*PRN*r*Tr+r*RHO*U
      DQ(MeqnW ,KderWz  ,ind) = (-4*LACPt*PRN*r*Tz)/3.d0+r*RHO*W
      DQ(MeqnW ,KderWrr ,ind) = -(LACP*PRN*r)
      DQ(MeqnW ,KderWzz ,ind) = (-4*LACP*PRN*r)/3.d0
      DQ(MeqnW ,KderTr  ,ind) = -(LACPt*PRN*r*Uz)-LACPt*PRN*r*Wr
      DQ(MeqnW ,KderTz  ,ind) = (2*LACPt*PRN*U)/3.d0                          &
     &                          +(2*LACPt*PRN*r*Ur)/3.d0                      &
     &                          -(4*LACPt*PRN*r*Wz)/3.d0

      DQ(MeqnT ,KderTr  ,ind) = -LACP-(2*CPt*LACP*r*Tr)/CP-2*LACPt*r*Tr       &
     &                          +r*RHO*U
      DQ(MeqnT ,KderTz  ,ind) = (-2*CPt*LACP*r*Tz)/CP-2*LACPt*r*Tz+r*RHO*W
      DQ(MeqnT ,KderTrr ,ind) = -(LACP*r)
      DQ(MeqnT ,KderTzz ,ind) = -(LACP*r)

      DQ(MeqnYK,KderPr  ,ind) = (r*RHO*UC*YK)/P
      DQ(MeqnYK,KderPz  ,ind) = (r*RHO*WC*YK)/P
      DQ(MeqnYK,KderTr  ,ind) = -((r*RHO*UC*YK)/T)-(LACPt*r*YKr)/LEK
      DQ(MeqnYK,KderTz  ,ind) = -((r*RHO*WC*YK)/T)-(LACPt*r*YKz)/LEK
      FORALL (k=1:NSP)
! roughly correct: dF(YK)/dYK is full since F(YK) contains RHO, a fn of all YK
      DQ(MeqnYK(k),KderYKr(k) ,ind) = -(LACP/LEK(k))-(LACPt*r*Tr)/LEK(k)      &
     &                                +r*RHO*U+r*RHO*UC                       &
     &                                +(r*RHO*UC*WM*YK(k))/WK(k)
      DQ(MeqnYK(k),KderYKz(k) ,ind) = -((LACPt*r*Tz)/LEK(k))+r*RHO*W          &
     &                                +r*RHO*WC+(r*RHO*WC*WM*YK(k))/WK(k)
      DQ(MeqnYK(k),KderYKrr(k),ind) = -((LACP*r)/LEK(k))
      DQ(MeqnYK(k),KderYKzz(k),ind) = -((LACP*r)/LEK(k))
      END FORALL
```

```
         IF (iDTIME.eq.3) THEN  ! Crank-Nicolson only

            DO k2 = 1, NVAR*NDRTP
            DO k1 = 1, NVAR
               DQ(k1,k2,ind) = 0.5d0*DQ(k1,k2,ind)
            ENDDO
            ENDDO

         ENDIF

      ENDDO
      ENDDO

!-------------------------------------------------------------------------------
!     JACOBIAN - BOUNDARY NODES
!-------------------------------------------------------------------------------

!     CONDITIONS AT THE INLET: BOUNDARY 1

      j = 1

      DO i = 2, NR

         ind = (j-1)*NR + i

         r   = RI(i)

         P0  = AtmPressure
         P2  = S(MvarP ,ind)
         P   = P0 + P2
!         P   = P0

         U   = S(MvarU ,ind)
         W   = S(MvarW ,ind)
         T   = S(MvarT ,ind)
         YK  = S(MvarYK,ind)

         Pr  = SD(KderPr  ,ind)
         Pz  = SD(KderPz  ,ind)
         Ur  = SD(KderUr  ,ind)
         Uz  = SD(KderUz  ,ind)
         Urr = SD(KderUrr ,ind)
         Uzz = SD(KderUzz ,ind)
         Urz = SD(KderUrz ,ind)
         Wr  = SD(KderWr  ,ind)
         Wz  = SD(KderWz  ,ind)
         Wrr = SD(KderWrr ,ind)
         Wzz = SD(KderWzz ,ind)
         Wrz = SD(KderWrz ,ind)
         Tr  = SD(KderTr  ,ind)
         Tz  = SD(KderTz  ,ind)
         Trr = SD(KderTrr ,ind)
         Tzz = SD(KderTzz ,ind)
```

```
!       time derivatives
        Wt    = 0.d0
        if (iDTIME.gt.1)                                                    &
     &     Wt = VELinlet(i)*MODULATION*2*PI/PERIOD*cos(timephase)

        WK    = MolecularWeights
        WM    = MixtureWeight (YK, WK)
        RHO   = P*MW/(RU*T)
        PDYN  = P2
        GZ    = GRAV

        CP      = SpecificHeat(T)
        CPt     = 0.d0               ! SpecificHeat_ddT(T)
        CPtt    = 0.d0               ! SpecificHeat_d2dT2(T)
        LACP    = LambdaCp(T)        ! lambda/Cp
        LACPt   = LambdaCp_ddT(T)    ! d(lambda/Cp)/dT
        LACPtt  = LambdaCp_d2dT2(T)  ! d2(lambda/Cp)/dT2
        PRN     = PrandtlNumber

        DJ(MeqnP ,MvarP ,ind) = -((GZ*r*RHO)/P)+(r*RHO*U*Wr)/P+(r*RHO*Wt)/P  &
     &                          +(r*RHO*W*Wz)/P
        DJ(MeqnP ,MvarU ,ind) = (2*LACPt*PRN*Tz)/3.d0+r*RHO*Wr
        DJ(MeqnP ,MvarW ,ind) = r*RHO*Wz
        ! from time derivative
        IF (iDTIME.gt.1)                                                    &
     &     DJ(MeqnP ,MvarW ,ind) = DJ(MeqnP ,MvarW ,ind)                    &
     &                             +(r*RHO*ddt)
        DJ(MeqnP ,MvarT ,ind) = (GZ*r*RHO)/T+(2*LACPtt*PRN*Tz*U)/3.d0       &
     &                          +(2*LACPtt*PRN*r*Tz*Ur)/3.d0                &
     &                          -(LACPt*PRN*r*Urz)/3.d0-(LACPt*PRN*Uz)/3.d0 &
     &                          -LACPtt*PRN*r*Tr*Uz-LACPt*PRN*Wr            &
     &                          -LACPtt*PRN*r*Tr*Wr-(r*RHO*U*Wr)/T          &
     &                          -LACPt*PRN*r*Wrr-(r*RHO*Wt)/T               &
     &                          -(4*LACPtt*PRN*r*Tz*Wz)/3.d0-(r*RHO*W*Wz)/T &
     &                          -(4*LACPt*PRN*r*Wzz)/3.d0
        DJ(MeqnP ,MvarYK,ind) = (GZ*r*RHO*WM)/WK-(r*RHO*U*WM*Wr)/WK          &
     &                          -(r*RHO*WM*Wt)/WK-(r*RHO*W*WM*Wz)/WK

        DJ(MeqnU ,MvarU ,ind) = 1

        DJ(MeqnW ,MvarW ,ind) = 1

        DJ(MeqnT ,MvarT ,ind) = 1

        FORALL (k=1:NSP)
        DJ(MeqnYK(k),MvarYK(k),ind) = 1
        END FORALL

        DQ(MeqnP ,KderPz  ,ind) = r
        DQ(MeqnP ,KderUr  ,ind) = (2*LACPt*PRN*r*Tz)/3.d0
        DQ(MeqnP ,KderUz  ,ind) = -(LACP*PRN)/3.d0-LACPt*PRN*r*Tr
        DQ(MeqnP ,KderUrz ,ind) = -(LACP*PRN*r)/3.d0
        DQ(MeqnP ,KderWr  ,ind) = -(LACP*PRN)-LACPt*PRN*r*Tr+r*RHO*U
        DQ(MeqnP ,KderWz  ,ind) = (-4*LACPt*PRN*r*Tz)/3.d0+r*RHO*W
        DQ(MeqnP ,KderWrr ,ind) = -(LACP*PRN*r)
        DQ(MeqnP ,KderWzz ,ind) = (-4*LACP*PRN*r)/3.d0
        DQ(MeqnP ,KderTr  ,ind) = -(LACPt*PRN*r*Uz)-LACPt*PRN*r*Wr
```

```
         DQ(MeqnP ,KderTz  ,ind) = (2*LACPt*PRN*U)/3.d0                    &
     &                             +(2*LACPt*PRN*r*Ur)/3.d0                &
     &                             -(4*LACPt*PRN*r*Wz)/3.d0

      ENDDO

!.....Special treatment at symmetry-inlet corner pt
      i   = 1
      ind = 1
      DJ(MeqnU  ,MvarU  ,ind) = 1
      DJ(MeqnW  ,MvarW  ,ind) = 1
      DJ(MeqnT  ,MvarT  ,ind) = 1
      FORALL (k=1:NSP)
      DJ(MeqnYK(k),MvarYK(k),ind) = 1
      END FORALL
      DQ(MeqnP ,KderPr  ,ind) = 1



                        <<<<< ... CUT ... >>>>>


!-------------------------------------------------------------------------------
!     SOME VARIABLES MAY BE FIXED
!-------------------------------------------------------------------------------

      if (KVFIX.gt.0) then
         do ind=1,NODES
            do k=1,NVAR
               if (NFIX(k).eq.1) then
                  do kk=1,NVAR
                     kkderind=(kk-1)*NDRTP
                     DJ(k,kk,ind) = 0.d0
                     DQ(k,kkderind+1,ind) = 0.d0
                     DQ(k,kkderind+2,ind) = 0.d0
                     DQ(k,kkderind+3,ind) = 0.d0
                     DQ(k,kkderind+4,ind) = 0.d0
                     DQ(k,kkderind+5,ind) = 0.d0
                  enddo
                  DJ(k,k,ind) = 1.d0
               endif
            enddo
         enddo
      endif


      RETURN
      END
```